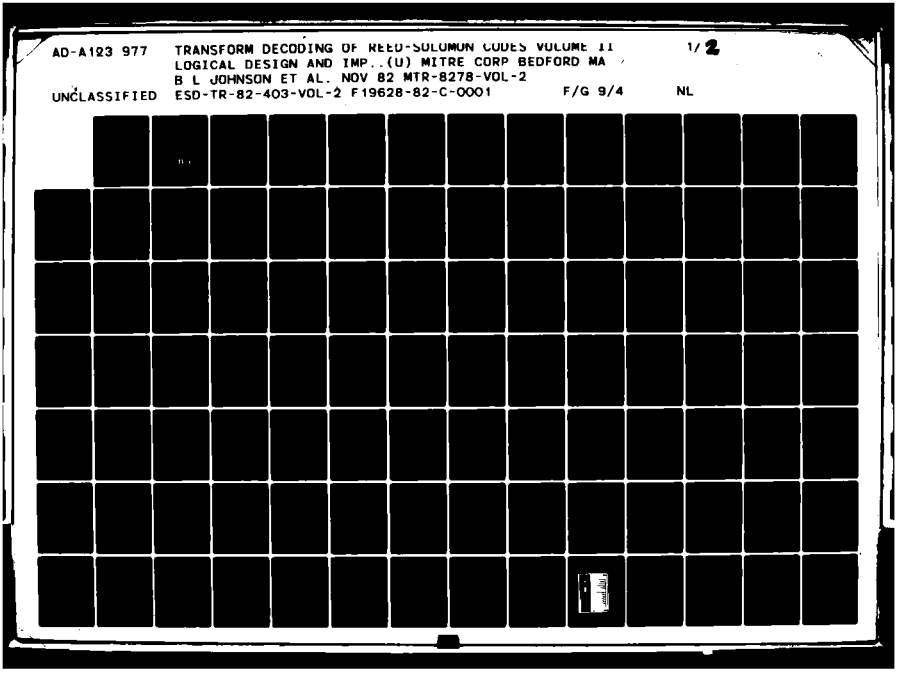
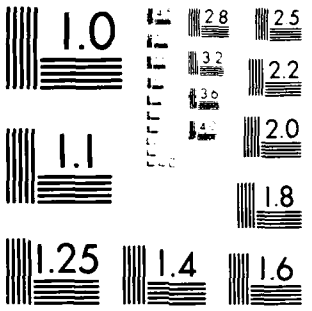
AD-A123 977 TRANSFORM DECODING OF REED-SOLOMON CODES VOLUME II 1/2
LOGICAL DESIGN AND IMP..(U) MITRE CORP BEDFORD MA
B L JOHNSON ET AL. NOV 82 MTR-8278-VOL-2
UNCLASSIFIED ESD-TR-82-403-VOL-2 F19628-82-C-0001 F/G 9/4 NL





MICROCOPY RESOLUTION TEST CHART

(12)

ESD-TR-82-403, Vol. II

MTR-8278, Vol. II

AD A123017

TRANSFORM DECODING OF REED-SOLOMON CODES VOLUME II: LOGICAL DESIGN AND IMPLEMENTATION

By
B. L. JOHNSON
A. L. BEQUILLARD
S. J. MEEHAN

NOVEMBER 1982

Prepared for

SOLID STATE SCIENCES DIVISION
ROME AIR DEVELOPMENT CENTER
UNITED STATES AIR FORCE
Hanscom Air Force Base, Massachusetts

DTIC
ELECTE
JAN 3 1 1983
S
B

DTIC FILE COPY

Approved for public release;
distribution unlimited

Project No. 7170
Prepared by
THE MITRE CORPORATION
Bedford, Massachusetts
Contract No. F19628-82-C-0001

When U S Government drawings, specifications, or other data are used for any purpose other than a definitely related government procurement operation, the government thereby incurs no responsibility nor any obligation whatsoever, and the fact that the government may have formulated, furnished, or in any way supplied the said drawings, specifications, or other data is not to be regarded by implication or otherwise, as in any manner licensing the holder or any other person or corporation, or conveying any rights or permission to manufacture, use, or sell any patented invention that may in any way be related thereto.

Do not return this copy. Retain or destroy.

REVIEW AND APPROVAL

This technical report has been reviewed and is approved for publication.

Jerry Silverman

JERRY SILVERMAN
Project Engineer

Harold Roth

HAROLD ROTH, Director
Solid State Sciences Division

UNCLASSIFIED

SECURITY CLASSIFICATION OF THIS PAGE (When Data Entered)

REPORT DOCUMENTATION PAGE		READ INSTRUCTIONS BEFORE COMPLETING FORM
1. REPORT NUMBER ESD-TR-82-403, Vol. II	2. GOVT ACCESSION NO.	3. RECIPIENT'S CATALOG NUMBER
4. TITLE (and Subtitle) TRANSFORM DECODING OF REED-SOLOMON CODES VOLUME II: LOGICAL DESIGN AND IMPLEMENTATION		5. TYPE OF REPORT & PERIOD COVERED 6. PERFORMING ORG. REPORT NUMBER MTR-8278, Vol. II
7. AUTHOR(s) B. L. JOHNSON, A. L. BEQUILLARD, S. J. MEEHAN		8. CONTRACT OR GRANT NUMBER(s) F19628-82-C-0001
9. PERFORMING ORGANIZATION NAME AND ADDRESS The MITRE Corporation Burlington Road Bedford, MA 01730		10. PROGRAM ELEMENT, PROJECT, TASK AREA & WORK UNIT NUMBERS Project No. 7170
11. CONTROLLING OFFICE NAME AND ADDRESS Solid State Sciences Division Rome Air Development Center Hanscom AFB, MA 01731		12. REPORT DATE November 1982 13. NUMBER OF PAGES 149
14. MONITORING AGENCY NAME & ADDRESS(if different from Controlling Office)		15. SECURITY CLASS. (of this report) UNCLASSIFIED 15a. DECLASSIFICATION DOWNGRADING SCHEDULE

16. DISTRIBUTION STATEMENT (of this Report)

Approved for public release; distribution unlimited.

17. DISTRIBUTION STATEMENT (of the abstract entered in Block 20, if different from Report)

18. SUPPLEMENTARY NOTES

19. KEY WORDS (Continue on reverse side if necessary and identify by block number)

ERROR LOCATION
LOGIC DESIGN
REED-SOLOMON CODES
SCHOTTKY TTL LOGIC
TRANSFORM ENCODING AND DECODING

20. ABSTRACT (Continue on reverse side if necessary and identify by block number)

This report describes the logic design and hardware implementation of an encoder and decoder for a large number of Reed-Solomon symbol error-correction codes. The logic implements a transform encoding and decoding algorithm that was previously described in Volume I of MTR-8278. The hardware required to implement the critical steps in the encoding and decoding algorithm is described in depth. An analysis of the decoder's operational characteristics and hardware complexity is presented. A proof-of-concept breadboard configured with small-scale Schottky TTL components is also described.

DD FORM 1 JAN 73 1473 EDITION OF 1 NOV 65 IS OBSOLETE

UNCLASSIFIED

SECURITY CLASSIFICATION OF THIS PAGE (When Data Entered)

ACKNOWLEDGMENTS

This document has been prepared under Project 7170, Contract F19628-82-C-0001. The contract is sponsored by the Solid State Sciences Division, Rome Air Development Center, Hanscom Air Force Base, Massachusetts.

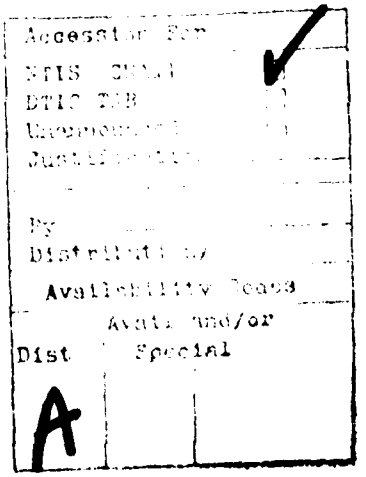


TABLE OF CONTENTS

TABLE OF CONTENTS (Continued)

TABLE OF CONTENTS (Concluded)

LIST OF ILLUSTRATIONS

LIST OF ILLUSTRATIONS (Concluded)

LIST OF TABLES

LIST OF TABLES (Concluded)

SECTION I

INTRODUCTION

1.1 Purpose

This report examines the hardware implementation of an efficient decoding algorithm for the Reed-Solomon class of symbol-error-correcting codes. The algorithm, described in Volume I of this report, offers major simplifications relative to the more conventional BCH (Bose-Chaudhuri-Hocquenghem) decoding algorithms [1]. The simplifications result both from the reduced complexity of the algorithm and from the opportunity to apply fast computational techniques for its implementation.

1.2 Background

Error-correcting codes are useful to correct message errors which are caused by interference, additive random noise, and other channel disturbances. Error-correction techniques are implemented in a two-step process. At the message source, redundant symbols are added to the original message according to a predetermined strategy (encoding). The encoded message is transmitted and errors may be introduced. At the message destination, the original message is recovered from the noisy received signal (decoding), aided by prior knowledge of the code. Message transmission using error-correcting codes represents an effective means of obtaining low error probability in the decoded message.

Previous work included both the analysis of error-correcting codes and the examination of their effective implementation. Error-correcting codes, used with spread-spectrum modulation techniques, were shown to be beneficial in the design of jam-resistant communica-

tion systems. It has also been suggested that error-correcting codes, when incorporated within the internal busing structure of a system (or device), can impact favorably on that system's (device's) reliability [2].

Our examination of error-correcting codes has led us to concentrate on the Reed-Solomon class of generalized BCH symbol error-correcting codes. The distance properties of this class of algebraic block codes assure correction of both random isolated errors and random burst errors. While the encoding process for Reed-Solomon codes is relatively simple, the decoding process is complex and generally requires a dedicated processor.

We have experimented with direct decoding of short block length Reed-Solomon codes by implementing a code-table search algorithm under microprocessor control [3]. Further analysis of Reed-Solomon codes has led to the development of a transform-based decoder that offers major simplifications relative to the more conventional BCH decoders [4].

The decoding algorithm imposes a high degree of circuit complexity on its associated hardware implementation. Analogies with conventional linear digital signal processing functions aid in partitioning the decoding hardware into sections that perform finite-field operations (e.g., field-element multiplication, division, and inversion). These sections can be used to develop functional LSI hardware which performs a variety of finite-field data processing functions. If the unique properties of finite structures are exploited (e.g., elimination of round-off errors, multiplication by adding "logarithms"), the development of these hardware capabilities may lead to the use of finite-field computational methods for other linear signal processing applications.

1.3 Scope

This is the second volume of a report concerned with transform decoding of Reed-Solomon codes. The first volume discussed the decoding algorithm. This volume concentrates on the logic design and hardware implementation of the transform decoding algorithm. It begins by outlining the concepts of transform coding and decoding of Reed-Solomon codes. Section II is primarily an overview of the material presented in Volume I, included here for completeness. While reading this report one should also refer to Volume I of this TR [4], which contains the appropriate frame of reference for the present volume.

In Section III, an architectural design of a Reed-Solomon encoder and decoder is presented. The processor is reconfigurable to accommodate a large number of different code parameters for both maximum and sub-maximum length codes over $GF(2^m)$, the symbol fields ranging from four to eight bits. The maximum-length codeword that can be processed by this design is a 255-symbol word, with each symbol represented by eight bits. (This unit will be called the (255,k) encoder and decoder.) Included within Section III is a detailed description of the coding capabilities, functional partitioning, projected hardware complexity and expected operational characteristics of the (255,k) transform encoder and decoder.

In Section IV, a description of the logic implementation of a reconfigurable Reed-Solomon TTL breadboard is presented. This encoder and decoder breadboard is designed to be electronically reconfigurable to accommodate a subset of the codes processed by the (255,k) encoder and decoder described in Section III. Although the breadboard is not large enough to decode all of the codes processed by the (255,k) decoder, it operates over most of the required fields and it effectively demonstrates the reconfigurability of the decoder's architecture. The encoder and decoder breadboard is capable

of processing a maximum-length codeword of 51 symbols, each symbol being represented by eight bits. (The breadboard will be called the (51,k) encoder and decoder.) Included within section IV is a detailed discussion of the breadboard's coding capabilities, functional and physical partitioning, hardware complexity and operational characteristics.

Appendix A presents a detailed discussion of binary-extension field multiplier structures that are used in the Reed-Solomon error-correcting encoder and decoder. The transform encoding and decoding of a (31,15) Reed-Solomon code, constructed over $GF(2^5)$, is presented by means of an example in appendix B to aid the reader in tracing the flow of the decoding algorithm.

SECTION II

TRANSFORM ENCODING AND DECODING: AN OVERVIEW

Reed-Solomon codes are symbol error-correcting linear block codes. A particular (n,k) Reed-Solomon code, constructed over the binary-extension field $GF(2^m)$, has a block length of n symbols, where k symbols (k<n) represent the information. Each of the n symbols within the codeword can be represented as a binary m-tuple.

Reed-Solomon codes are maximum-distance separable linear block codes. These are (n,k) codes for which the minimum distance, d_{min}, between any pair of codewords is the maximum value,

$$d_{min} = n - k + 1 \tag{2-1}$$

These codes can correct any combination of t errors and s erasures provided the inequality

$$2t + s \leq n - k \tag{2-2}$$

is satisfied.

In order to discuss the structural properties of Reed-Solomon codes and their implementation, it is convenient to regard the codewords as polynomials. A codeword from an (n,k) code, constructed over $GF(2^m)$, is an n-tuple with each symbol represented by m bits. Each codeword can be represented by a polynomial of degree n-1, having coefficients that are members of the finite field of 2^m elements.

Such a polynomial is determined uniquely by its n coefficients or equivalently by its values at any n distinct points of the field. A codeword of block length n may be specified either by a set of n values or by the polynomial coefficients interpolated from these values.

2.1 Finite Field Transforms Over $GF(2^m)$: A Review

Let $a_0, a_1, \ldots, a_{n-1}$ be elements of a finite field $GF(2^m)$ of multiplicative order 2^m-1. Let b be an element of $GF(2^m)$, and let b be an n^{th} root of unity. Assuming that n divides or is equal to 2^m-1, the linear transformation

$$A_j = \sum_{i=0}^{n-1} a_i b^{ij} \quad ; j = 0, 1, \ldots, n-1 \qquad (2\text{-}3)$$

is a mapping from $GF(2^m)$ onto itself. For any integer r,

$$\sum_{i=0}^{n-1} b^{ir} = \begin{cases} n, & r \equiv 0 \bmod n \\ 0, & \text{otherwise} \end{cases} \qquad (2\text{-}4)$$

Equation (2-4) can be used to verify that the mapping that is inverse to equation (2-3) is the linear transformation

$$a_i = n^{-1} \sum_{j=0}^{n-1} A_j b^{-ji} \quad ; i=0, 1, \ldots, n-1 \qquad (2\text{-}5)$$

where $n^{-1}n = 1$. Equations (2-3) and (2-5) define a discrete linear transform pair over $GF(2^m)$, where the operations of addition and multiplication are defined in the same field. Addition of two field elements from $GF(2^m)$ is defined as the bit-by-bit modulo-2 addition

of the m-tuple representation of the field elements. Multiplication is defined in terms of the primitive field element α. $GF(2^m)$ has multiplicative order 2^m-1 and it contains an element α of the same order. The non-zero elements of the field can be written as $\alpha^0, \alpha^1, \alpha^2, \ldots, \alpha^{2^m-2}$. Multiplication of two field elements is defined as the addition (modulo 2^m-1) of the indices of the corresponding field elements

$$\alpha^r \cdot \alpha^s = \alpha^{(r+s)} \tag{2-6}$$

The sequence $a_o, a_1, \ldots, a_{n-1}$ of elements from the field $GF(2^m)$ can be expressed as an (n-1)th degree polynomial, a(z), where

$$a(z) = a_0 + a_1 z + \ldots + a_{n-1} z^{n-1} = \sum_{i=0}^{n-1} a_i z^i \tag{2-7}$$

The forward transform of a sequence $a_0, a_1, a_2, \ldots, a_{n-1}$ can be obtained by the polynomial evaluation of a(z) at the n distinct powers of the transform's kernel; $b^0, b^1, \ldots, b^{n-1}$, such that

$$A_j = \sum_{i=0}^{n-1} a_i b^{ij} = a(b^j) \quad ; \quad j=0, 1, \ldots, n-1 \tag{2-8}$$

Similarly, the inverse transform is obtained by interpolation of the polynomial a(z) from its n known values.

$$a_i = \sum_{j=0}^{n-1} A_j \, b^{-ji} = A(b^{-j}) \; ; \; i=0, 1, \ldots, n-1 \tag{2-9}$$

where $A(z) = A_0 + A_1 z + \ldots + A_{n-1} z^{n-1}$.

2.2 Codeword Generation by Discrete Transformation

The encoding of Reed-Solomon codes can be defined in terms of finite-field transforms. Let $a_0, a_1, \ldots, a_{k-1}$ represent a sequence of k message symbols, with each symbol represented by m bits. A length-n message sequence can be formed by adjoining n-k consecutive zero-valued symbols to the original length-k message sequence. We regard the polynomial a(z) as the message polynomial

$$a(z) = a_0 + a_1 z + \ldots + a_{n-1} z^{n-1} \qquad (2\text{-}10)$$

The first k coefficients are the k message symbols, and the remaining n-k coefficients are zero.

A codeword for a Reed-Solomon (n,k) code, constructed over $GF(2^m)$, can be generated by calculating the n-point discrete transform of the sequence represented by a(z), [4,5]. There is symmetry associated with the transform so that either a forward or an inverse transform may be used to encode. The only requirement is that the reverse of the encoding transform be calculated for decoding. For compatibility with Volume I, this review will use a forward transform for encoding. A codeword, consisting of n symbols, is constructed by calculating the forward transform of the length-n message sequence, as in equation (2-8). The forward transform, or polynomial evaluation, can be expressed as the continued product

$$A_j = a_0 + b^j(a_1 + \ldots + b^j (a_{n-2} + b^j a_{n-1})\ldots) \qquad (2\text{-}11)$$

Equivalently, the forward transform can be interpreted as the remainder of the polynomial division $a(z)/(z-b^j)$

$$a(z) = q(z)(z-b^j) + A_j \qquad (2\text{-}12)$$

The second interpretation may be represented as a set of polynomial congruences such that

$$A_j = a(b^j) \equiv a(z) \bmod (z-b^j) \;\; ; \;\; j=0, 1, \ldots, n-1 \qquad (2\text{-}13)$$

An equivalent method for obtaining A_j is to divide $a(z)$ by a set of small degree polynomials containing distinct factors of the form $(z-b^j)$, and then to evaluate the lesser-degree residue polynomials at the appropriate values b^j. If the set of divisor polynomials is defined to be the set of minimal polynomials of the nonzero field elements, then their coefficients are restricted to the prime field GF(2). In this case, division can be performed using only scalar multiplication by the elements of the prime field. For codes over $GF(2^m)$, the minimal polynomials have coefficients that are either one or zero requiring only operations in GF(2) for polynomial division. In Volume I, this technique of computing a finite-field transform was shown to be a "fast" algorithm; it tends to minimize the number of multiplications in $GF(2^m)$, the number approaching $n\log_2 n$.

2.3 Reed-Solomon Transform Decoding

Finite-field transforms can be applied to decode Reed-Solomon codewords. If the source message is represented by the polynomial expressed in equation (2-10), then the transmitted codeword is represented by the polynomial $A(z) = A_0 + A_1 z + \ldots + A_{n-1} z^{n-1}$ where the coefficients A_j are determined as $a(b^j)$ in accordance with equation (2-8). If the inverse finite-field transform, equation (2-5), is applied to the transmitted codeword, the message polynomial a(z) is obtained and the original k message symbols are recovered.

Assume that an error sequence represented by the polynomial $E(z) = E_0 + E_1 z + \ldots + E_{n-1} z^{n-1}$ has been added to the encoded message A(z) during transmission. In order for the received word to be correctable, E(z) can not have more than (n-k)/2 non-zero coefficients; their values and locations are unknown. The received sequence is represented by the polynomial sum R(z) = E(z) + A(z). The inverse transform of the received sequence is the polynomial sum r(z) = e(z) + a(z), where e(z) is the inverse transform of the error polynomial E(z), and a(z) is the original length-n message polynomial. The decoding problem is to determine e(z) from the transform r(z) of the observed sequence R(z).

To decode, the polynomial r(z) is calculated from the known values of the received sequence R(z) by taking its inverse transform,

$$r_i = \sum_{j=0}^{n-1} R_j b^{-ij} \qquad i=0, 1, \ldots, n-1 \qquad (2\text{-}14)$$

which is equivalent to evaluating the received polynomial R(z) at the n values, $b^0, b^{-1}, \ldots, b^{-(n-1)}$.

The symbols a_i, $i > k-1$, are equal to zero by definition. A

sequence can be separated from equation (2-14), valid for i=k, k+1, ..., n-1:

$$s_i = r_i = \sum_{j=0}^{n-1} R_j b^{-ij} \qquad i=k,\ k+1,\ \ldots,\ n-1 \qquad (2\text{-}15)$$

This sequence, $\{s_i\}$, is the error syndrome associated with the channel error pattern, E(z).

The error syndrome can be used to determine the locations of the errors in the channel error pattern E(z), using the iterative algorithm developed by Berlekamp and Massey [6,7]. This algorithm calculates the coefficients of the error-locator polynomial,

$$\sigma(z) = \prod_{i=1}^{t} (z - X_i) = \sigma_t + \sigma_{t-1} z + \ldots + z^t \qquad (2\text{-}16)$$

whose distinct roots X_i are the error locations. In equation (2-16) t is the number of non-zero coefficients of E(z), or equivalently the number of errors that occurred. We assume $t \leq (n-k)/2$ so that the error bound of the code is not exceeded. The error-locator polynomial is the characteristic polynomial of the shortest linear feedback shift register (LFSR) that satisfies uniquely a linear recursion relationship between the n-k syndrome values and the coefficients of the error-locator polynomial.

$$s_j \sigma_t + s_{j+1} \sigma_{t-1} + \ldots + s_{j+t-1} \sigma_1 + s_{j+t} = 0 \qquad (2\text{-}17)$$

where $k \leq j \leq n-1-t$.

The Berlekamp-Massey algorithm uses as its inputs the error syndrome values and provides an iterative method for synthesizing the shortest LFSR that has the characteristic polynomial $\sigma(z)$. Once the LFSR has been synthesized by this algorithm, it is necessary only to continue its operation, with zero input, for an additional k shifts in order to extrapolate the k unknown values of the error transform e(z). These values are subtracted from the corresponding value of r(z) in order to produce the corrected message, a(z).

2.3.1 Correction of Errors and Erasures

The previously described decoding algorithm has been concerned only with correcting errors. A Reed-Solomon code can correct twice as many erasures as errors; it can correct any pattern of t errors and s erasures provided the inequality of equation (2-2) is satisfied. A useful Reed-Solomon decoder should be capable of correcting both errors and erasures.

A method of correction for errors and erasures is to initialize the error-locator algorithm (Berlekamp-Massey) with the connection polynomial computed from the known erasure locations, and then continue the algorithm normally to synthesize an errata-locator polynomial which is the product of the error-locator polynomial and the erasure-locator polynomial [4]. Once the errata-locator polynomial is synthesized, there is no further distinction between errors and erasures, and the inverse transform of the errata pattern may be extrapolated by free-running the synthesized LFSR as before. These values are then subtracted from the corresponding values of r(z) in order to decode the correct message.

The erasure-locator polynomial, $\lambda(z)$, is defined as

$$\lambda(z) = \prod_{i=1}^{s} (z-\tilde{X}_i) = \lambda_s + \lambda_{s-1}z + \ldots + z^s \qquad (2\text{-}18)$$

where s erasures have occurred, not exceeding the minimum-distance bound of equation (2-2). The roots, $\tilde{X}_i$, designate the known erasure locations forming a set that is disjoint from the error locations, X_i. The modified Berlekamp-Massey algorithm iteratively calculates the errata-locator polynomial, $\tilde{\sigma}(z)$, which is the product of the error-locator and erasure-locator polynomials:

$$\tilde{\sigma}(z) = \sigma(z)\ \lambda(z) \tag{2-19}$$

The errata-locator polynomial is then used to generate the transform of the channel errata pattern which is subtracted from the transform of the received data to obtain the decoded message.

2.4 Transform Encoding and Decoding: Hardware Structures

The preceding view concerning the transform encoding and decoding of Reed-Solomon codes was meant to be general in nature. The required computational steps and procedures do not imply uniqueness of hardware implementation. For example, transform codeword generation requires that n-k consecutive zeros be padded to the original k information symbols in order to form the length-n message sequence. In section 2.2, the zero-padding was defined so that the k information symbols, and the remaining n-k coefficients were zero. This zero-padding placement is not unique; the cyclic properties of the code result in many possible zero-padding placements. Each results in a slightly different design and physical implementation for the transform encoder and decoder, without modifying the general algorithm.

There is also symmetry associated with transform encoding and decoding. A forward transform may be defined for encoding; an inverse transform would then be required for decoding. Alternately, an inverse transform may be defined for encoding and a forward transform for decoding. Either approach is correct: their hardware implementations differ.

Regardless of the particular variations, all transform-based encoders and decoders will have common characteristics. A representative block diagram of a communication system that uses transform error-correction encoding and decoding techniques is shown in Figure 1. The first step in encoding requires that the k information symbols be padded to n symbols with n-k zeros. The second step in encoding is the calculation of the n-point discrete forward (or inverse) linear transformation. These n symbols are then transmitted and corrupted by noise in the channel. At the receiver, the noisy symbols are observed and the symbols that are erasures are identified. The received symbols and the locations of the known erasures are sent to the decoder. The decoder first calculates the required n-point discrete inverse (or forward) linear transformation. The known erasure locations are used to initialize the errata-location section with the erasure-locator polynomial. The n-k syndrome values are separated from the transform of the received symbols and are used as inputs to the errata locator. This section calculates the errata-locator polynomial as in equation (2-19). The errata-location section then calculates the transform of the errata that occur during transmission. This data is subtracted from the transform of the received data, recovering the k original information symbols. The total number of errors and erasures is assumed to be within the bound of the code given in equation (2-2).

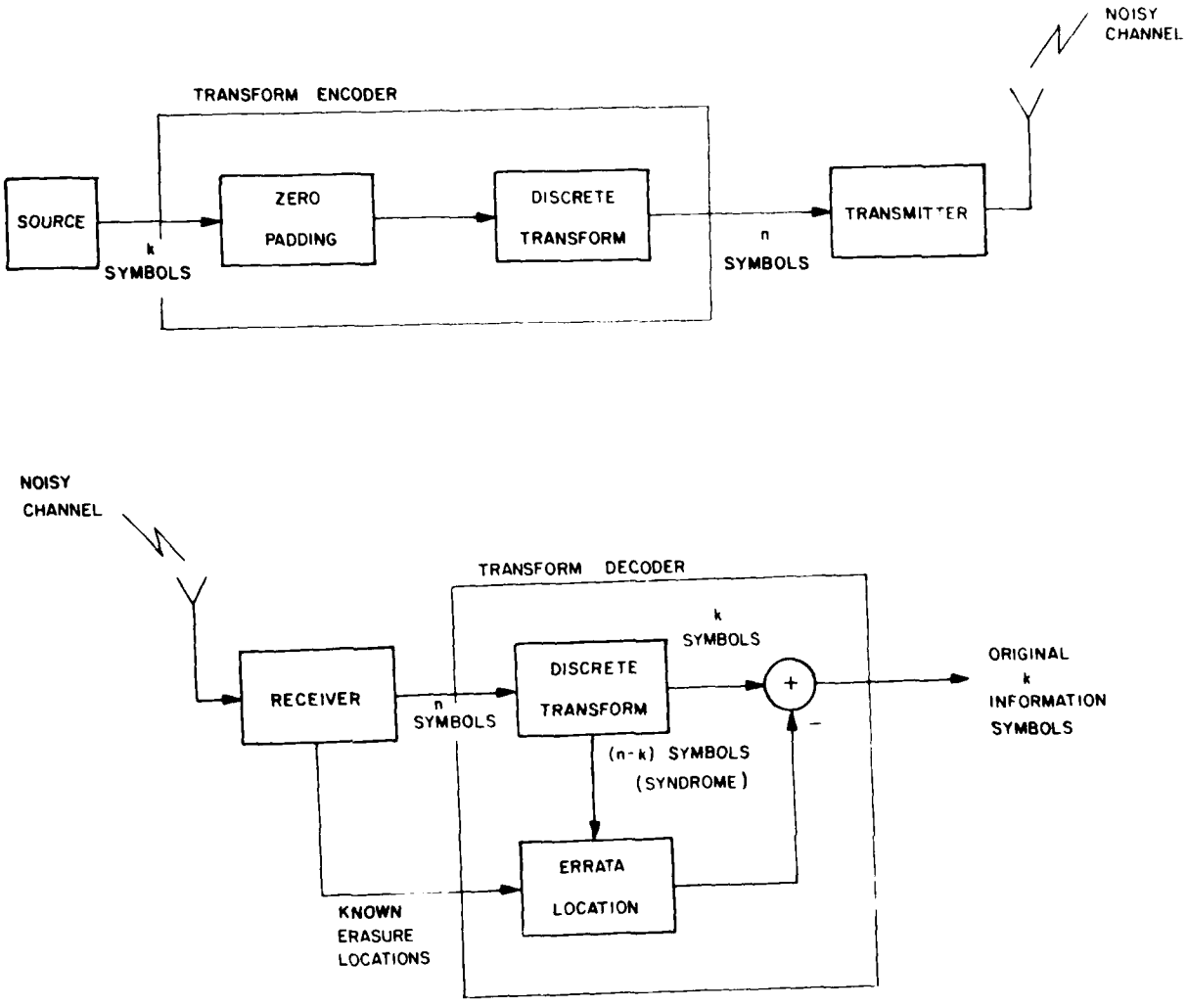


Figure 1. Transform Encoding and Decoding

SECTION III

A (255,k) REED-SOLOMON TRANSFORM ENCODER AND DECODER

A design at the detailed logic level of a transform encoder and decoder for use with the Reed-Solomon class of symbol error-correcting codes is described in this section. It is a computationally efficient implementation of the transform decoding algorithm described in Volume I of this report (and summarized in section II of this volume). The encoder and decoder can implement a 255-symbol block-length code, as well as many shorter codes. It is designated as the (255,k) encoder and decoder.

3.1 General Description

A useful error-correcting encoder and decoder should operate with a number of different code parameters in order to be applicable to various channel characteristics and system designs. The error controller's hardware implementatior must be capable of implementing different block lengths and different symbol alphabets. To encode and decode an (n,k) code constructed over $GF(2^m)$, the hardware must implement an n-point finite-field transform where each symbol in the transform is represented by m bits. The ability to calculate transforms of different lengths over different finite fields requires that the hardware be able to implement algebraic operations that are defined in the different fields. The essential algebraic operations that must be implemented are field-element addition, multiplication, and inversion. Field-element multiplication is defined uniquely for each binary-extension field, and the hardware that implements multiplication in one field must be reconfigured to multiply correctly in another. (See appendix A of this report for a more detailed description of $GF(2^m)$ multiplier structures.) In general the implementation

of a versatile encoder and decoder requires hardware reconfigurability to operate successfully in different binary-extension fields.

Our implementation of the transform decoding algorithm was designed to minimize the total number of binary-extension-field multiplications required for both encoding and decoding [4]. The (255,k) encoder and decoder was designed to operate with serial input code symbols so that many of the required finite-field multiplications can be calculated sequentially in time using the same hardware. The resulting architecture tends to minimize the total number of $GF(2^m)$ multipliers that have to be implemented, minimizing the amount of hardware reconfigurability and the resulting hardware complexity required to accommodate the codes from the different binary-extension fields.

The natural partitioning of the transform decoding algorithm separates the decoder's structure into a transform section, an errata-location section, and a control section. The encoding algorithm partitions the encoder into a transform section and a control section. We developed the logical design of a general transformer that implements a computationally efficient number-theoretic transform algorithm [10]. The transform section was designed to calculate both a forward and an inverse discrete transform over the fields of interest. The same structure can be used for both encoding and decoding, resulting in a considerable saving in hardware design and fabrication, thus rendering it suitable for a VLSI chip-set implementation.

The control section provides data management to the transform and errata-location sections. This control can be implemented using standard TTL logic or dedicated LSI or VLSI circuitry. Control also could be provided by use of a software-programmable microprocessor. This report is not concerned further with the detailed design of the control section. The architectural design of the transform and errata-location sections, which carry out the major computational steps in the encoding and decoding algorithms, are emphasized in this section.

The hardware complexity associated with both the transform section and the errata-location section is such that each section could be implemented using a single VLSI monolithic device [11]. This level of complexity is fundamental to the concept of a versatile encoder and decoder. Since the same transformer can be used for computing either a forward or an inverse discrete transform, a complete encoder and decoder can be implemented using only two devices. A transform "chip" and an errata-location "chip" would be required for decoding, while only the transform "chip" would be required for encoding. The hardware necessary to perform encoding is inherently contained within the hardware required for decoding.

3.1.1 Coding Capabilities

The (255,k) Reed-Solomon encoder and decoder was designed to provide a selection of useful codes while containing the complexity of the projected hardware. The range of Reed-Solomon codes that can be processed by the (255,k) encoder and decoder design is shown in Table I. These codes represent a large number of both maximum and submaximum length codes over $GF(2^m)$ where the symbol representation, m, ranges from four to eight bits.

The errata-location section's architecture is bit-slice and expandable to accommodate any code rate; each symbol used for redundancy requires a corresponding hardware slice within the decoder. However, it is desirable that the errata-locator be implementable as a single integrated circuit, and this requirement restricts the errata-locator's implementation to a size (total number of transistors) that can process a maximum of 128 symbols used for redundancy. Since Reed-Solomon codes are maximum distance separable codes, this restricts the largest value of d_{min} to 129 and equivalently restricts the largest number of syndrome symbols to 128. The (255,k) decoder is consequently designed to operate with a maximum of

Table I

Reed-Solomon (255,k) Encoder and Decoder Capabilities

MAXIMUM BLOCK LENGTH (SYMBOLS)	BITS/SYMBOL m	(n, k) CODES $(n-k) \leq 128$	
255 (17 x 5 x 3)	8	(255, k) (85, k) (51, k) (17, k) (15, k) (5, k) (3, k)	304 Total Codes
127 (PRIME)	7	(127, k)	127 Total
63 (7 x 3 x 3)	6	(63, k) (21, k) (9, k) (7, k) (3, k)	103 Total
31 (PRIME)	5	(31, k)	31 Total
15 (3 x 5)	4	(15, k) (5, k) (3, k)	23 Total

of 128 syndrome symbols, regardless of code block size. For a Reed-Solomon (n,k) code, the number of syndrome symbols is (n-k). Accordingly, the (255,k) decoder can correct all combinations of t errors, and s erasures, provided the ineqality

$$2t + s \leq n-k \leq 128 \qquad (3\text{-}1)$$

is satisfied.

The Reed-Solomon (255,k) encoder and decoder design can accommodate 588 distinct codes defined by different allowed choices of the parameters n and k. This number is derived from the maximum number of syndrome symbols and the variety of code classes that can be processed. For example, the (255,k) class of codes, constructed over $GF(2^8)$, represents a family of codes whose block length is fixed at 255 symbols but whose number of information symbols, k, is a variable. The design trade-offs which restrict the maximum number of allowable syndrome symbols to 128 define a total of 128 distinct codes in this class (i.e., k can range from 127 to 255). For the other families of (n,k) codes shown in Table I, k can range from 1 to n. Some of these codes are trivial but most are not. Table II indicates the seventeen approximately half-rate codes that can be accommodated by the (255,k) encoder and decoder's design.

3.2 (255,k) Transform Encoder and Decoder Architecture

The (255,k) encoder and decoder is partitioned into a transform section and an errata-location section. The transform section implements either a forward or an inverse n-point discrete transform and it is used for either encoding or decoding. The errata-location section implements a modified version of the Berlekamp-Massey minimal-length LFSR synthesis algorithm. This algorithm, used for decoding, corrects erasures as well as errors. Both the transform section and the errata-

Table II

Half-Rate Codes Accommodated by the (255,k)
Reed-Solomon Transform Decoder

CODE (n,k)	BITS PER SYMBOL m
(255, 127)	8
(127, 63)	7
(85, 42)	8
(63, 31)	6
(51, 25)	8
(31, 15)	5
(21, 10)	6
(17, 8)	8
(15, 7)	8, 4
(9, 4)	6
(7, 3)	6
(5, 2)	8, 4
(3, 1)	8, 4, 6

location section are reconfigurable to operate over the binary-extension fields, $GF(2^m)$, with m ranging from four to eight bits.

3.2.1 Transform Section

The range of transforms that can be calculated by the transform section of the (255,k) encoder and decoder is shown in the Table III. This table indicates the number of symbols in the transform, n, the number of bits per symbol, m, (specifying the field of operation $GF(2^m)$), and the kernel of the transform, α^K. For an n-point transform over $GF(2^m)$, the kernel is an n^{th} root of unity, that is, α^K is an element of $GF(2^m)$ of multiplicative order n, so that n is the least integer for which $\alpha^{Kn} = 1$.

To calculate an n-point forward transform over $GF(2^m)$ the transform section must implement

$$A_j = \sum_{i=0}^{n-1} a_i \alpha^{Kij} \quad ; \quad j=0, 1, \ldots, n-1 \qquad (3\text{-}2)$$

where,

$$A_j, a_i \in GF(2^m) \quad ; \quad 0 \leq i, j \leq n-1$$

and $\alpha^K \in GF(2^m)$, with multiplicative order n.

Table III

Transform Capabilities of (255,k) Decoder's Transform Section

Transform Size n	Bits Per Symbol m	Kernel of Transform α^K
255	8	α^1
127	7	α^1
85	8	α^3
63	6	α^1
51	8	α^5
31	5	α^1
21	6	α^3
17	8	α^{15}
15	8	α^{17}
15	4	α^1
9	6	α^7
7	6	α^9
5	8	α^{51}
5	4	α^3
3	8	α^{85}
3	6	α^{21}
3	4	α^5

As mentioned in section II, a forward transform can be interpreted as polynomial evaluation, which can be represented as

$$A_j = a(\alpha^{Kj}) \quad ; \quad j=0, 1, \ldots, n-1 \tag{3-3}$$

where a(x) is an (n-1)th degree polynomial over $GF(2^m)$.

To calculate an inverse n-point finite-field transform, over $GF(2^m)$, the transform section must implement:

$$a_i = \sum_{j=0}^{n-1} A_j \, \alpha^{-Kij} \quad ; \quad i=0, 1, \ldots, n-1 \tag{3-4}$$

or equivalently,

$$a_i = A(\alpha^{-Ki}) \quad ; \quad i=0, 1, \ldots, n-1 \tag{3-5}$$

where A(z) is an (n-1)th degree polynomial defined over $GF(2^m)$.

To calculate either a forward or an inverse finite-field transform the transform section implements a polynomial evaluation algorithm. To calculate a forward n-point transform, over $GF(2^m)$, the transform section evaluates an (n-1)th degree polynomial at the n distinct powers of the element α^K. To calculate an n-point inverse

finite-field transform over the same field, the transform section still evaluates an (n-1)th degree polynomial at the n distinct powers of the element α^K, but the order of evaluation is reversed since $\alpha^{-K\ell} = \alpha^{K(n-\ell)}$.

The transform section implements a computationally efficient algorithm for polynomial evaluation. For an n-point transform, the n points to be transformed are defined as the coefficients of an (n-1)th degree polynomial over $GF(2^m)$. This data polynomial is divided by a set of polynomials of smaller degree whose roots are conjugate sets of the n distinct powers of α^K. Each remainder polynomial, or residue polynomial, is then evaluated at each of the conjugate roots of its corresponding divisor polynomial in order to obtain the transformed points. The set of divisor polynomials is the same for either a forward or an inverse transform; the order of evaluation determines which transform is being calculated.

An n-point transform pair is defined on $GF(2^m)$ if n divides 2^m-1. If n equals 2^m-1, then the transform is maximum-length, the set of divisor polynomials is the set of all minimal polynomials associated with $GF(2^m)$, and the n points of evaluation are the 2^m-1 non-zero field elements. If V is greater than one, where $Kn = 2^m-1$, then the transform is submaximum-length and the set of divisor polynomials is defined as the set of minimal polynomials that have the n distinct powers of α^K as roots. The points at which the residue polynomials are evaluated are the n powers of α^K.

In order to evaluate an ℓ-th degree remainder polynomial at the point α^j, the following equation is implemented.

$$r(\alpha^j) = r_0 + r_1(\alpha^j) + r_2(\alpha^j)^2 + \cdots + r_\ell(\alpha^j)^\ell \qquad (3\text{-}6)$$

This is equivalent to a continued product expansion

$$r(\alpha^j) = (\cdots(r_\ell\alpha^j + r_{\ell-1})\alpha^j + \cdots + r_1)\alpha^j + r_0 \qquad (3\text{-}7)$$

This expansion can be effectively implemented using an extension field multiplier and accumulator as a polynomial evaluator. The symbol errata-locator requires one transformed symbol at a time, and the transform section is required to supply sequentially-calculated transform points. A single polynomial-evaluator circuit may be multiplexed to calculate the desired n transform symbols.

The operation of the transform section can be partitioned into two functions. The transformer first divides the (n-1)th degree data polynomial simultaneously by all minimal polynomials of the elements of $GF(2^m)$. Then, each point in the transform is sequentially calculated by evaluating the appropriate residue polynomial at the corresponding element in the field. The order of evaluation determines whether the transform is forward or inverse. A block diagram of the transform section is shown in Figure 2. In this figure, the transform section is partitioned into a polynomial residue calculator, a multiplexer, a polynomial residue evaluator, and an arithmetic controller.

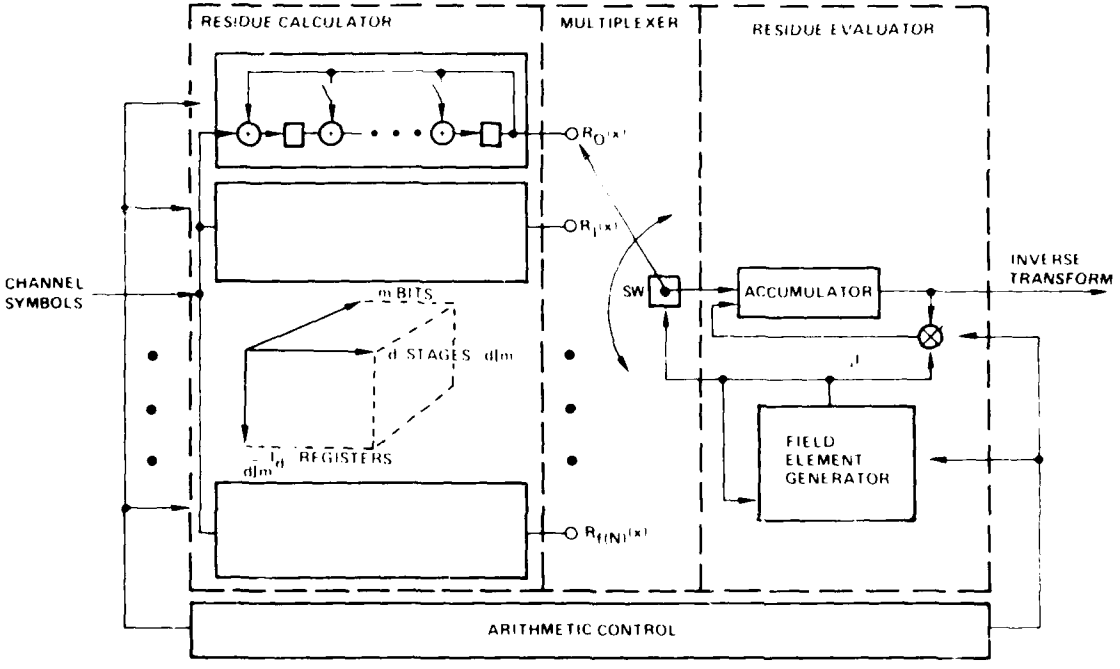


Figure 2. Transform Section Architecture

3.2.1.1 Polynomial Residue Calculator. As a first step in calculating an n-point transform over $GF(2^m)$ the polynomial residue calculator simultaneously divides the polynomial representing the data to be transformed by all minimal polynomials of the elements of $GF(2^m)$. Polynomial division is implemented with LFSRs whose feedback-connection polynomials are defined to be the divisor polynomials. The fast polynomial evaluation algorithm defines the divisor polynomials to be the minimal irreducible polynomials from the finite field of operation. This means that the divisor polynomials for the residue calculators are irreducible over GF(2) and the coefficients of the corresponding LFSR's feedback connection polynomials are restricted to either one or zero. Therefore, there are no extension-field multiplications required to implement the division portion of the fast polynomial evaluation algorithm. Division can be implemented using only scalar multiplication (by either zero or one) and GF(2) (modulo-two) addition.

The polynomial residue calculator is capable of dividing by all the minimal polynomials in $GF(2^m)$, where m=4,5,6,7, and 8. A complete list of these polynomials is presented in Tables IV-1 through IV-3. There are a total of 66 polynomials for the five different binary-extension fields. In order to provide all of the transform capabilities indicated in Table III, the polynomial residue calculator must be capable of dividing by all 66 minimal polynomials; however, only simultaneous division by the polynomials from the field of operation is required for calculating a particular transform. The binary-extension field $GF(2^8)$ has the largest number of minimal polynomials; there are 35 divider circuits to be implemented for transformation in this field. The key to minimizing the residue calculator's hardware is to design these 35 circuits to be reconfigurable in order to provide for division by the remaining 31 minimal polynomials needed for transformation in the four other finite fields.

To facilitate the description of the residue calculator's

Table IV - 1

Minimal Irreducible Polynomials over $GF(2^8)$

$$m_i(z) = m_0 + m_1 z^1 + m_2 z^2 + m_3 z^3 + m_4 z^4 + m_5 z^5 + m_6 z^6 + m_7 z^7 + m_8 z^8$$

Polynomials	m_o	m_1	m_2	m_3	m_4	m_5	m_6	m_7	m_8
m_0 (z)	1	1	0	0	0	0	0	0	0
m_1 (z)	1	0	1	1	1	0	0	0	1
m_3 (z)	1	1	1	0	1	1	1	0	1
m_5 (z)	1	1	0	0	1	1	1	1	1
m_7 (z)	1	0	0	1	0	1	1	0	1
m_9 (z)	1	0	1	1	1	1	0	1	1
m_{11} (z)	1	1	1	0	0	1	1	1	1
m_{13} (z)	1	1	0	1	0	1	0	0	1
m_{15} (z)	1	1	1	0	1	0	1	1	1
m_{17} (z)	1	1	0	0	1	0	0	0	0
m_{19} (z)	1	0	1	0	0	1	1	0	1
m_{21} (z)	1	1	0	1	0	0	0	1	1
m_{23} (z)	1	1	0	0	0	1	1	0	1
m_{25} (z)	1	1	0	1	1	0	0	0	1
m_{27} (z)	1	1	1	1	1	1	0	0	1
m_{29} (z)	1	0	1	1	0	0	0	1	1
m_{31} (z)	1	0	1	1	0	1	0	0	1
m_{37} (z)	1	1	1	1	1	0	1	0	1
m_{39} (z)	1	0	0	1	1	1	1	1	1
m_{43} (z)	1	1	0	0	0	0	1	1	1
m_{45} (z)	1	0	0	1	1	1	0	0	1
m_{47} (z)	1	0	0	1	0	1	0	1	1
m_{51} (z)	1	1	1	1	1	0	0	0	0
m_{53} (z)	1	1	1	0	0	0	0	1	1
m_{55} (z)	1	0	0	0	1	1	0	1	1
m_{59} (z)	1	0	1	1	0	0	1	0	1
m_{61} (z)	1	1	1	1	0	0	1	1	1
m_{63} (z)	1	0	1	1	1	0	1	1	1
m_{85} (z)	1	1	1	0	0	0	0	0	0
m_{87} (z)	1	1	0	0	0	1	0	1	1
m_{91} (z)	1	0	1	0	1	1	1	1	1
m_{95} (z)	1	1	1	1	1	0	0	1	1
m_{111}(z)	1	1	0	1	1	1	1	0	1
m_{119}(z)	1	0	0	1	1	0	0	0	0
m_{127}(z)	1	0	0	0	1	1	1	0	1

Table IV - 2

Minimal Irreducible Polynomials over $GF(2^7)$

$$m_i(z) = m_0 + m_1 z^1 + m_2 z^2 + m_3 z^3 + m_4 z^4 + m_5 z^5 + m_6 z^6 + m_7 z^7$$

Polynomial	m_0	m_1	m_2	m_3	m_4	m_5	m_6	m_7
m_0 (z)	1	1	0	0	0	0	0	0
m_1 (z)	1	0	0	1	0	0	0	1
m_3 (z)	1	1	1	1	0	0	0	1
m_5 (z)	1	0	1	1	1	0	0	1
m_7 (z)	1	1	1	0	1	1	1	1
m_9 (z)	1	1	1	1	1	1	0	1
m_{11} (z)	1	0	1	0	1	0	1	1
m_{13} (z)	1	1	0	0	0	0	0	1
m_{15} (z)	1	1	1	1	0	1	1	1
m_{19} (z)	1	1	0	1	0	0	1	1
m_{21} (z)	1	0	1	0	0	1	1	1
m_{23} (z)	1	0	0	0	0	0	1	1
m_{27} (z)	1	1	0	0	1	0	1	1
m_{29} (z)	1	1	0	1	0	1	0	1
m_{31} (z)	1	0	0	0	1	1	1	1
m_{43} (z)	1	1	1	0	0	1	0	1
m_{47} (z)	1	0	0	1	1	1	0	1
m_{55} (z)	1	0	1	1	1	1	1	1
m_{63} (z)	1	0	0	0	1	0	0	1

Minimal Irreducible Polynomials over $GF(2^6)$

$$m_i(z) = m_0 + m_1 z^1 + m_2 z^2 + m_3 z^3 + m_4 z^4 + m_5 z^5 + m_6 z^6$$

Polynomial	m_0	m_1	m_2	m_3	m_4	m_5	m_6
m_0	1	1	0	0	0	0	0
m_1	1	1	0	0	0	0	1
m_3	1	1	1	0	1	0	1
m_5	1	1	1	0	0	1	1
m_7	1	0	0	1	0	0	1
m_9	1	0	1	1	0	0	0
m_{11}	1	0	1	1	0	1	1
m_{13}	1	1	0	1	1	0	1
m_{15}	1	0	1	0	1	1	1
m_{21}	1	1	1	0	0	0	0
m_{23}	1	1	0	0	1	1	0
m_{27}	1	1	0	1	0	0	0
m_{31}	1	0	0	0	0	1	1

Table IV - 3

Minimal Irreducible Polynomials over $GF(2^5)$

$$m_i(z) = m_0 + m_1 z^1 + m_2 z^2 + m_3 z^3 + m_4 z^4 + m_5 z^5$$

Polynomial	m_0	m_1	m_2	m_3	m_4	m_5
m_0	1	1	0	0	0	0
m_1	1	0	1	0	0	1
m_3	1	0	1	1	1	1
m_5	1	1	1	0	1	1
m_7	1	1	1	1	0	1
m_{11}	1	1	0	1	1	1
m_{15}	1	0	0	1	0	1

Minimal Irreducible Polynomials over $GF(2^4)$

$$m_i(z) = m_0 + m_1 z^1 + m_2 z^2 + m_3 z^3 + m_4 z^4$$

Polynomial	m_0	m_1	m_2	m_3	m_4
m_0	1	1	0	0	0
m_1	1	1	0	0	1
m_3	1	1	1	1	1
m_5	1	1	1	0	0
m_7	1	0	0	1	1

architecture, it is advantageous to examine the structure used to implement polynomial division. Figure 3 shows the structure of an LFSR that is used to perform polynomial division. (A detailed description of the operation of this circuit can be found in Chapter 7 of reference [1].) The circuit shown is designed to divide by the polynomial $M_{95} = x^8 + x^7 + x^4 + x^3 + x^2 + x + 1$, which is a minimal polynomial from $GF(2^8)$. The positions of the feedback taps are determined by the coefficients of the divisor polynomial. In order to perform division, the registers are all cleared to zero, and the data representing the polynomial to be divided is fed sequentially into the shift register. After the last symbol is entered, the remainder polynomial, or residue polynomial, is stored in the registers of the divider circuit. This residue, $R(x) = R_0 + R_1x + \ldots + R_7x^7$, is required for completion of the fast polynomial evaluation algorithm.

The structure shown in Figure 3 is designed to operate with symbols from $GF(2^8)$: the data lines are eight wide, the delay stages are eight registers deep, and the Exclusive-OR circuits operate with eight-bit words. Since the divisor polynomial contains only either zero or one as coefficients, the circuit shown in Figure 3 can be interpreted as eight identical binary feedback shift registers (BFSRs), each circuit capable of accommodating a single bit of each eight-bit input symbol. Each of the eight BFSRs contains delay stages that are only one bit deep, and the modulo-two adders are two-input binary Exclusive-OR gates, forming eight identical "slices", each physically separate from its seven companions.

All 35 minimal polynomials from $GF(2^8)$ can be implemented using circuits that are similar to the structure shown in Figure 3. In order to implement the polynomial division in $GF(2^8)$, a total of eight identical slices of hardware is required for each polynomial. Each slice contains 35 different BFSRs with each shift register having a maximum length of eight stages.

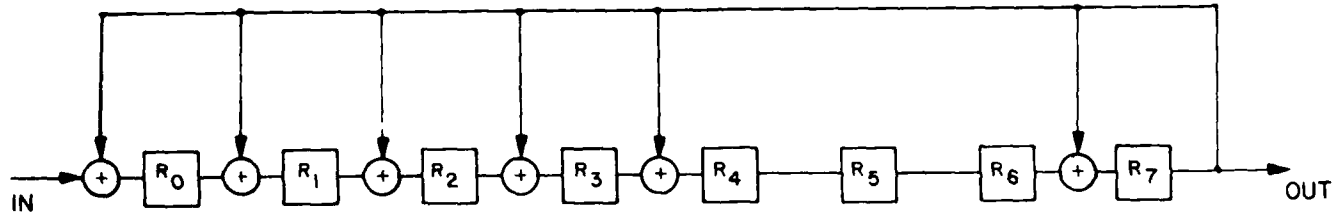


Figure 3. Polynomial Divider Circuit for
$M_{95}(x) = x^8 + x^7 + x^4 + x^3 + x^2 + x + 1$

There are two fundamental problems associated with designing the 35 divider circuits required for operation in $GF(2^8)$ to be reconfigurable to operate in the other finite fields. First of all, the different fields of operation have symbol sizes that range from four to eight bits, and the divider circuits must be capable of operating with these different symbol sizes. Secondly, the circuits must be reconfigurable to provide for division by different divisor polynomials. The positions of the feedback taps as well as the lengths of the registers are subject to reconfigurability. Both problems are made more difficult because of the desire to design the divider circuits to be as versatile as possible, but we would also like to keep the total amount of hardware at a reasonable level without incurring a large overhead for reconfigurability.

The necessity to operate with different symbol sizes is a consideration that recurs throughout the design of both the transform section and errata-location section. Our approach is to define a standard symbol size of eight bits and design all hardware to accommodate this symbol size and to be programmable for smaller fields. Since the hardware must accommodate symbols from $GF(2^8)$, no additional hardware is required when defining an eight-bit standard symbol, but some overhead is incurred for reconfigurability.

Any symbol from the field $GF(2^m)$ can be represented as an m bit sequence

$$\alpha^k \in GF(2^m) \leftrightarrow \{\alpha_0^k, \alpha_1^k, \cdots, \alpha_{m-1}^k\}$$

where (3-8)

$$\alpha_i^k \in GF(2) \quad ; \quad i=0, 1, \ldots, m-1.$$

Any symbol from $GF(2^m)$, where $m \leq 8$, can be represented as a binary eight-tuple where some of the bits are set to zero. We have defined our standard symbol as an eight-bit word such that α^k is represented as

$$\alpha^k \in GF(2^m)$$

$$\alpha^k = \{\underbrace{\alpha_0^k, \alpha_1^k, \ldots, \alpha_{m-1}^k}_{m} \underbrace{0, \ldots, 0}_{8-m}\} \qquad (3\text{-}9)$$

The notation of equation (3-9) will be frequently referred to as our "standard" symbol in this report.

When the residue calculator is operating with symbols from $GF(2^m)$, where $m<8$, our definition of standard symbol size results in zeros being fed into the 8-m slices corresponding to the bit-positions greater than m-1. The operation of the BFSRs associated with these zeros has no effect on the m slices required for the desired division.

The problem of designing the 35 divider circuits to be reconfigurable to provide division by all necessary minimal polynomials reduces to the problem of designing 35 BFSRs to be reconfigurable for the required division. The design can then be repeated eight times to obtain the parallel structure for the eight-bit polynomial residue calculator.

A goal associated with the design of a reconfigurable-divider circuit is to minimize the amount of hardware required for programmability. The design must be reconfigurable to accommodate different divisor polynomials of varying length. Minimizing the amount of hardware required for reconfigurability is roughly equivalent to minimizing the number of programmable feedback taps. Each programmable feed-

back tap allows the connection of a shift register's output to the particular stage where the tap is located. The hardware associated with a programmable tap must be repeated on all eight slices, and accompanying discrete logic or memory must be dedicated to control the operation of each programmable tap.

Two techniques can be combined to minimize the amount of hardware required for programming the BFSRs. Both techniques exploit the fact that the maximum degree of a minimal polynomial from $GF(2^m)$ is m. The divider circuits are designed originally to implement simultaneous division by the 35 minimal polynomials in $GF(2^8)$. A subset of these circuits is required to be reconfigurable in order to implement division by the 19 minimal polynomials in $GF(2^7)$. A second subset of the original circuit is required to be reconfigurable for operation in $GF(2^6)$. A third and fourth subset are required for operation in $GF(2^5)$ and $GF(2^4)$. Each of the four subsets requires BFSRs that are shorter in length then the eight stages required for operation in $GF(2^8)$.

The first minimization design technique is to group minimal polynomials from different fields that have identical or similar coefficients. Programmable taps are only required where discrepancies between tap weights occur. The second minimization technique is to design the output tap of each shift register to be programmable so that division by polynomials of different degrees can be implemented in the same circuit.

An illustrative example helps to clarify these concepts. Figure 4-a shows the logic level design of a BFSR that represents a single slice of a divider circuit suitable for use in the polynomial residue calculator. The circuit consists of an eight-stage feedback shift register whose output tap can be selected from one of five locations. The output multiplexer selects the position of the output tap that defines the feedback connection polynomial (divisor polynomial). The reconfigurability of this circuit for division in the different fields is shown in Table V.

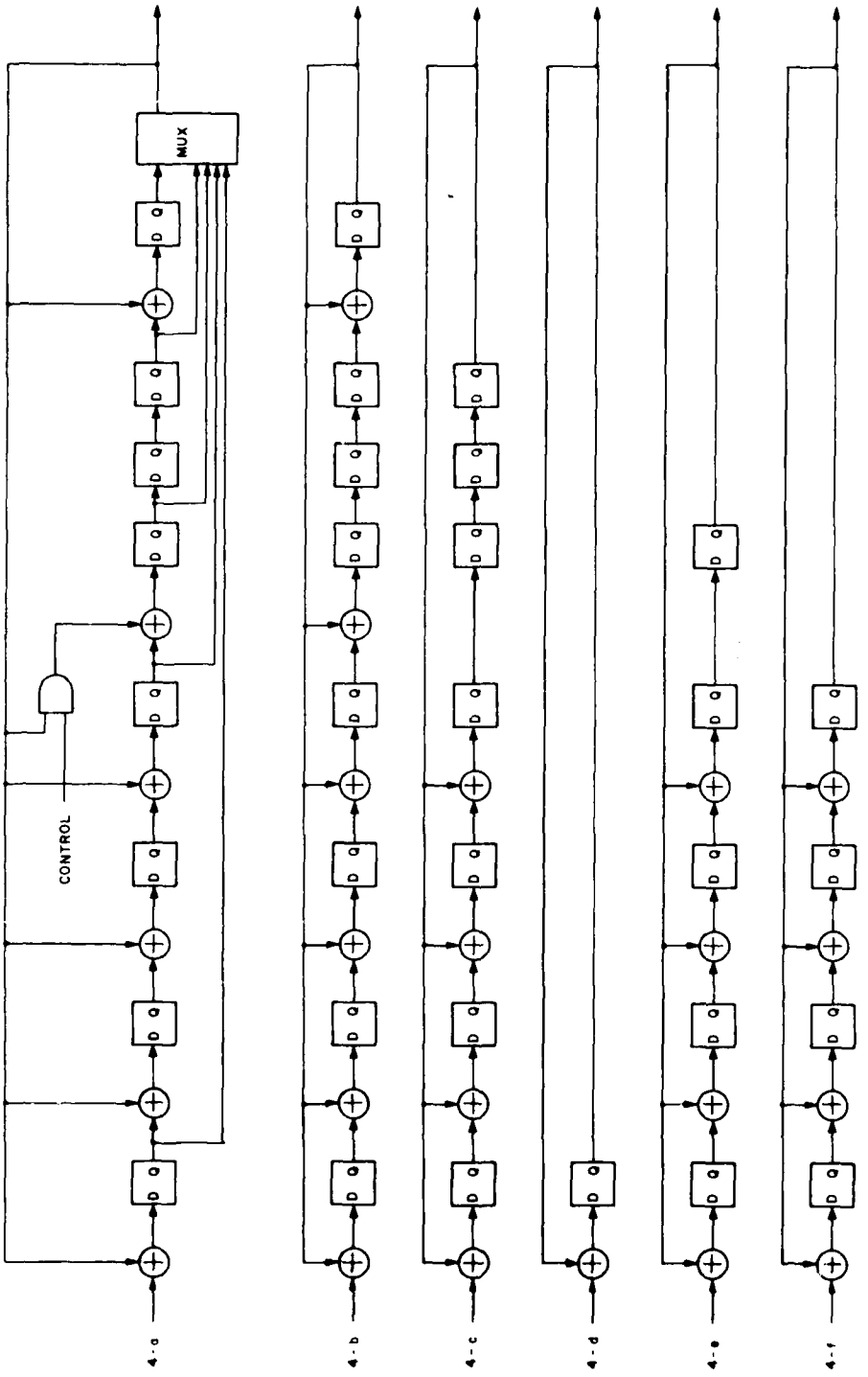


Figure 4. Programmable Binary Feedback Shift Register

TABLE V

Programmability of Binary Feedback Shift Register: Figure 4

Figure	Field of Operation	Divisor Polynomial
4-b	$GF(2^8)$	$M_{95}(x) = 1 + x + x^2 + x^3 + x^4 + x^7 + x^8$
4-c	$GF(2^7)$	$M_3(x) = 1 + x + x^2 + x^3 + x^7$
4-d	$GF(2^6)$	$M_0(x) = 1 + x$
4-e	$GF(2^5)$	$M_7(x) = 1 + x + x^2 + x^3 + x^5$
4-f	$GF(2^4)$	$M_3(x) = 1 + x + x^2 + x^3 + x^4$

Each of the original 35 divider circuits can be designed in a manner similar to that shown in Figure 4. Unfortunately, there is no readily apparent systematic method for assigning subsets and feedback taps. However, the design methodology results in hardware-efficient structures. The (51,k) breadboard to be described in section IV was designed to accommodate a large subset of the codes that can be processed by the (255,k) encoder and decoder. The breadboard's polynomial residue calculator was designed using the minimization techniques described in this section, and only three programmable taps were required, one on each of three separate divider circuits.

The fundamental structure of the polynomial residue calculator consists of eight identical slices of hardware. Each slice consists of 35 BFSRs, each being reconfigurable to provide for division by a set of different-length divisor polynomials. The maximum-length

divisor polynomial that can be implemented has degree eight; consequently, the maximum length of any feedback shift register is eight. The polynomial to be divided is fed sequentially into all 35 divider circuits. Division is completed after the last coefficient of the data polynomial has been entered. At this time, the calculated residue polynomials are stored within the delay stages of the divider circuits. These residue polynomials are transferred into a temporary holding memory, and the divider circuits are available for processing the next block of data. The residue polynomials are then available in temporary memory for further processing for completion of the fast polynomial evaluation algorithm. In this manner, the polynomial residue calculator can be thought of as pipelined, capable of simultaneously operating on two contiguous blocks of n symbols, thus accepting a continuous input stream.

3.2.1.2 Polynomial Residue Evaluator. The polynomial residue evaluator implements the second portion of the polynomial-evaluation algorithm. For operation in $GF(2^m)$ (regardless of code block size), the polynomial residue calculator provides the residue evaluator with the remainder polynomials that result from the division of the input data sequence by all the minimal polynomials in $GF(2^m)$. The residue evaluator sequentially calculates each point in the n-point transformation by selecting a predetermined residue polynomial and evaluating that residue at a root of its corresponding divisor polynomial. By definition, the point of evaluation is a power of α^K. If the transform is of maximum length, $n=2^m-1$, then the residue associated with each minimal polynomial in the field will be used at least once. If the transform is submaximum, $n|2^m-1$, then the residues associated with a subset of the minimal polynomials from $GF(2^m)$ will be used. Table VI indicates the subsets of minimal polynomials associated with each of the transforms listed in Table III.

The only difference between the computation of a forward and an inverse n-point transform over $GF(2^m)$ is the order in which the trans-

Table VI

Transforms over $GF(2^m)$

Field Of Calculation	Transform Length N	Required Minimal Polynomial Divisors
$GF(2^8)$	255	$m_0(z)$, $m_1(z)$, $m_3(z)$, $m_5(z)$, $m_7(z)$, $m_9(z)$, $m_{11}(z)$, $m_{13}(z)$, $m_{15}(z)$, $m_{17}(z)$, $m_{19}(z)$, $m_{21}(z)$, $m_{23}(z)$, $m_{25}(z)$, $m_{27}(z)$, $m_{29}(z)$, $m_{31}(z)$, $m_{37}(z)$, $m_{39}(z)$, $m_{43}(z)$, $m_{45}(z)$, $m_{47}(z)$, $m_{51}(z)$, $m_{53}(z)$, $m_{55}(z)$, $m_{59}(z)$, $m_{61}(z)$, $m_{63}(z)$, $m_{85}(z)$, $m_{87}(z)$, $m_{91}(z)$, $m_{95}(z)$, $m_{111}(z)$, $m_{119}(z)$, $m_{127}(z)$,
	85	$m_0(z)$, $m_3(z)$, $m_9(z)$, $m_{15}(z)$, $m_{21}(z)$, $m_{27}(z)$, $m_{39}(z)$, $m_{45}(z)$, $m_{51}(z)$, $m_{63}(z)$, $m_{87}(z)$, $m_{111}(z)$,
	51	$m_0(z)$, $m_5(z)$, $m_{15}(z)$, $m_{25}(z)$, $m_{45}(z)$, $m_{55}(z)$, $m_{85}(z)$, $m_{95}(z)$,
	17	$m_0(z)$, $m_{15}(z)$, $m_{45}(z)$
	15	$m_0(z)$, $m_{17}(z)$, $m_{51}(z)$, $m_{85}(z)$, $m_{119}(z)$
	5	$m_0(z)$, $m_{51}(z)$
	3	$m_0(z)$, $m_{85}(z)$
$GF(2^7)$	127	$m_0(z)$, $m_1(z)$, $m_3(z)$, $m_5(z)$, $m_7(z)$, $m_9(z)$, $m_{11}(z)$, $m_{13}(z)$, $m_{15}(z)$, $m_{19}(z)$, $m_{21}(z)$, $m_{23}(z)$, $m_{27}(z)$, $m_{29}(z)$, $m_{31}(z)$, $m_{43}(z)$, $m_{47}(z)$, $m_{55}(z)$, $m_{63}(z)$,

Table VI (Concluded)

Transforms over $GF(2^m)$

Field of Calculation	Transform Length N	Required Minimal Polynomial Divisors
$GF(2^6)$	63	$m_0(z)$, $m_1(z)$, $m_3(z)$, $m_5(z)$, $m_7(z)$, $m_9(z)$, $m_{11}(z)$, $m_{13}(z)$, $m_{15}(z)$, $m_{21}(z)$, $m_{23}(z)$, $m_{27}(z)$, $m_{31}(z)$
	21	$m_0(z)$, $m_3(z)$, $m_9(z)$, $m_{15}(z)$, $m_{21}(z)$, $m_{27}(z)$
	9	$m_0(z)$, $m_7(z)$, $m_{21}(z)$
	7	$m_0(z)$, $m_9(z)$, $m_{27}(z)$
	3	$m_0(z)$, $m_{21}(z)$
$GF(2^5)$	31	$m_0(z)$, $m_1(z)$, $m_3(z)$, $m_5(z)$, $m_7(z)$, $m_{11}(z)$, $m_{15}(z)$
$GF(2^4)$	15	$m_0(z)$, $m_1(z)$, $m_3(z)$, $m_5(z)$, $m_7(z)$
	5	$m_0(z)$, $m_3(z)$
	3	$m_0(z)$, $m_5(z)$

former symbols are calculated. This is a bookkeeping matter irrelevant to the architecture and operation of the residue calculator. The order of evaluation is determined by the arithmetic controller. For each transform point, the controller (see section 3.2.1.3) provides the residue evaluator with all the information needed to implement the polynomial evaluation algorithm.

A block diagram of the polynomial residue evaluator is shown in Figure 5. For each point in the transform, the polynomial residue evaluator performs two major operations. First, the input multiplexer selects a residue polynomial from the residue calculator. Secondly, the residue evaluator calculates each point in the transform by evaluating the selected residue using a multiplier and accumulator defined for $GF(2^m)$. The central components of the polynomial residue evaluator are the $GF(2^m)$ multiplier and accumulator. The remainder of this section will concentrate on a description of their design and operation.

The residue evaluator implements polynomial evaluation using the continued product expansion of equation (3-7). The expansion is well-suited for sequential implementation using the $GF(2^m)$ multiplier and accumulator shown in Figure 5. In order to compute all the transforms shown in Table III, the residue evaluator must be capable of operation in all five binary extension fields. Using our standard symbol notation, equation (3-9), the accumulator is easily implemented using eight two-input Exclusive-OR gates. (The padding of zeros for fields with $m<8$ automatically produces the correct results.) The multiplier structure chosen for use in the residue evaluator is an asynchronous array $GF(2^m)$ multiplier (described in appendix A of this report). This multiplier is the processing bottleneck within the transform section, and the array-type structure offers the fastest multiplication rates. However, a penalty is paid for this speed because the hardware implementation of the array-type structure requires the maximum number of gates of all $GF(2^m)$ multiplier structure alternatives, but note that the $GF(2^m)$ multiplier required for

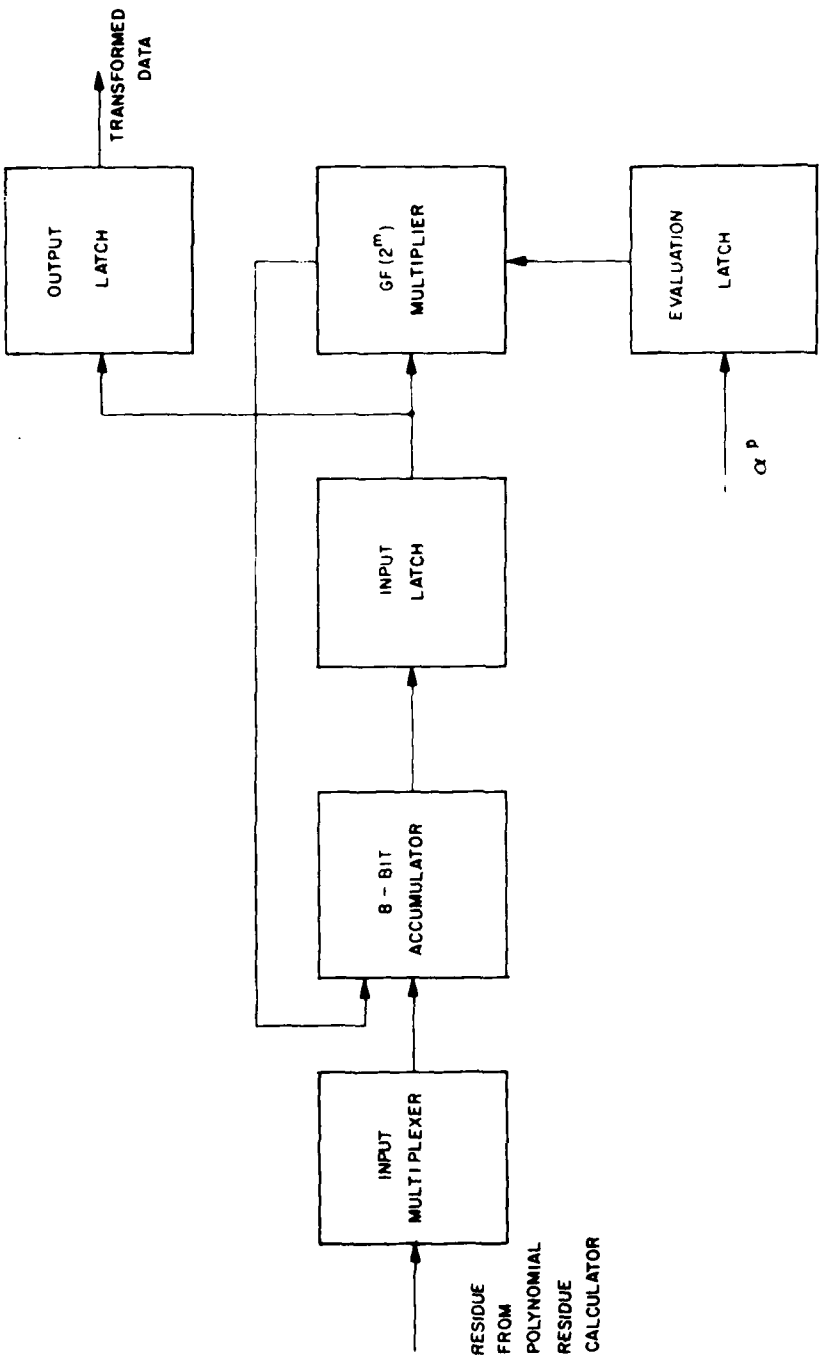


Figure 5. Polynomial Residue Evaluator

the residue evaluator section is used to perform the only extension-field multiplications in the transform architecture. Also, it will be shown that exactly the same $GF(2^m)$ multiplier structure is used to implement a critical portion of the errata-location section. The careful analysis and design of this multiplier results in a functional module whose usefulness overshadows the disadvantages associated with its complexity.

The operation of the polynomial residue evaluator can be described with an example. To calculate a point in the transform, the arithmetic controller supplies the polynomial residue calculator with (1) the necessary information to obtain the predetermined residue polynomial, (2) the degree of that particular residue polynomial, ℓ, and (3) the power of the kernel, $\alpha^{Ki} = \alpha^{p}$, at which the residue is to be evaluated. The residue evaluator is required to implement the expression

$$R(\alpha^{p}) = R_0 + R_1\alpha^{p} + \ldots + R_{\ell-1}\alpha^{(\ell-1)p} \qquad (3\text{-}10)$$

The coefficients of the residue polynomial are stored in a temporary memory within the residue calculator. The information provided by the arithmetic controller selects the appropriate memory locations, and serially feeds these coefficients, most significant coefficient first, into the multiplier and accumulator circuitry. The input latch (see Figure 5) is initially cleared to zero, and therefore the output of the programmable $GF(2^m)$ multiplier is also zero. The point of evaluation, α^{p}, is latched into the evaluation latch. The most significant coefficient of the residue polynomial is fed unchanged through the accumulator and latched in the input latch. After processing delay, the output of the asynchronous multiplier is $(R_{\ell-1}\alpha^{p})$.

This output is fed back into one input of the accumulator. Simultaneously, the next most significant coefficient is retrieved from the temporary memory and fed to the accumulator. The next output of the accumulator, $(R_{\ell-1}\alpha^p + R_{\ell-2})$, is held in the input latch and asynchronously multiplied with α^p. This process continues $(\ell-1)$ times until the input latch contains

$$(\cdots(R_{\ell-1}\alpha^p + R_{\ell-2})\alpha^p + \cdots R_1)\alpha^p + R_0 = R(\alpha^p) \qquad (3\text{-}11)$$

which is the evaluated polynomial. This data, or transformed symbol, is latched into the residue evaluator's output latch where it can be shifted out of the transform section for further processing. The entire process is repeated n times in order to compute an n-point transform. After each symbol is calculated, the input latch (Figure 5) must be cleared to zero.

3.2.1.3 Arithmetic Controller. The arithmetic controller provides all timing and control signals required to operate the transform section. As mentioned previously, the calculations implemented by both the polynomial residue calculator and polynomial residue evaluator are independent of whether a forward or an inverse transformation is performed. The arithmetic controller determines the order of the evaluation and therefore dictates the type of transform to be computed.

The arithmetic controller requires specific input data to provide management for the transformer. The controller needs to know whether the transformer is to be used for encoding (forward transform) or decoding (inverse transform). Also, the controller needs to know which code is being processed and the field in which the code is defined. From this information, the arithmetic controller generates the order of computation for the transform and its kernel.

The arithmetic controller provides minimal control to the polynomial residue calculator. For operation in a particular field, the controller reconfigures the LFSRs to divide by all the minimal polynomials within that field. The controller selects the shift register's output tap (via the output multiplexer) for each divider circuit and the controller also programs the necessary feedback taps. The controls for the polynomial residue calculator are static; once the code and field of operation are defined, the circuits are programmed and they remain unchanged for the duration of the transform calculation.

Primarily, the arithmetic controller manages the polynomial residue evaluator. Once the field of operation is defined, the controller reconfigures the $GF(2^m)$ multiplier to operate in that field. The multiplier remains in this configuration for the duration of the transform calculation. However, for each point in the transform, the controller must supply the residue calculator with the data required to select the predetermined residue polynomial. The controller must also provide the degree of that residue as well as the point of evaluation. These sets of control signals are dynamic; they change for the calculation of each transform point.

The arithmetic controller could be implemented with any of a number of hardware structures, including microprocessor controlled hardware. However, high throughput in the transformer warrants a high-speed controller. The controller architecture that is described in the following paragraphs was implemented in the (51,k) encoder and decoder breadboard.

The major function that must be implemented by the controller is the dynamic generation of the data required to calculate each individual transformed symbol. The static control required for each code can be easily generated with discrete combinational logic. A block diagram of an arithmetic controller is shown in Figure 6. The

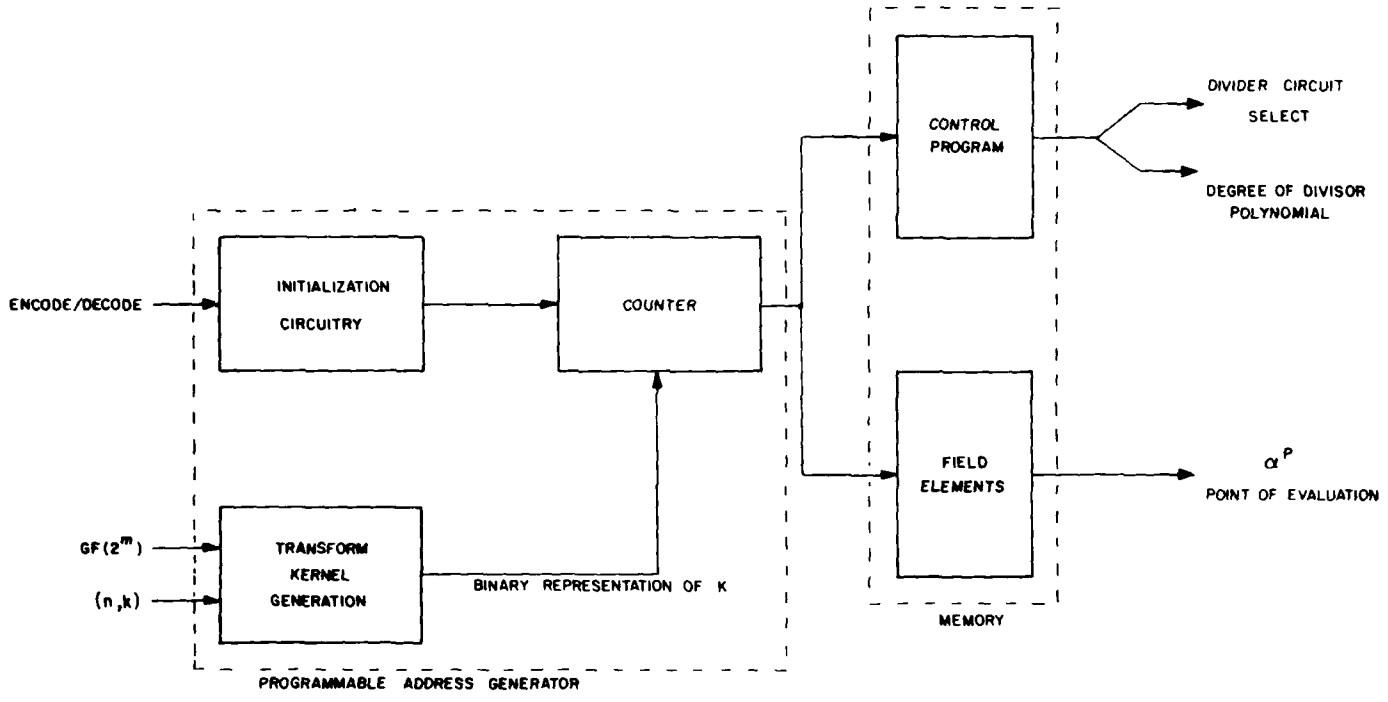


Figure 6. Arithmetic Controller

inputs for the controller are an encode/decode signal, the field of operation $GF(2^m)$, and the chosen code parameters (n,k). For each transformed symbol, the arithmetic controller generates three pieces of information: the address used by the residue evaluator's input multiplexer to select the predetermined residue polynomial, the degree of that residue polynomial, and the field element at which the polynomial is to be evaluated. The controller consists of preprogrammed memory and a programmable memory address generator. The memory is partitioned into two separate storage areas that have a common address. One memory section contains the field elements associated with each field and the other contains the information required to select and evaluate the residue polynomials. Within a given field, a particular element is a root of only one minimal polynomial. Therefore, for polynomial evaluation there exists a one-to-one relationship between any field element and its associated residue polynomial. When a field element is selected in one memory section, the data associated with its residue polynomial is selected in the other memory.

There are 2^m-1 nonzero field elements in the field $GF(2^m)$. These elements can be designated as $\alpha^0, \alpha^1, \alpha^2, \ldots, \alpha^{2^m-2}$. Our eight-bit standard symbol representation (equation (3-9)) of each element, α^i, from $GF(2^m)$ is stored in memory location 2^m+i. The field element α^0 from $GF(2^m)$ is an important evaluation point. It is stored in memory location 2^m and it is also stored in the memory location $2^m + 2^m - 1$ or $2^{m+1} - 1$. The residue-polynomial information corresponding to each field element is similarly stored in the second memory.

The programmable memory address generator consists of an initialization circuit, a transform kernel generator, and a programmable up-down counter. For a forward n-point transform over $GF(2^m)$, the field elements required for evaluation in the fast polynomial algorithm are $(\alpha^0, \alpha^K, \alpha^{2K}, \ldots, \alpha^{(n-1)K})$. The memory address corresponding

to these field elements is generated by initializing the up-down counter to the memory location 2^m, and then incrementing the counter by K for each point in the transform. For an inverse n-point transform over $GF(2^m)$, the field elements required for evaluation are $(\alpha^0, \alpha^{(n-1)K}, \alpha^{(n-2)K}, \ldots, \alpha^{2K}, \alpha^K)$. The memory address corresponding to these elements are generated by initializing the up-down counter to memory location $2^{m+1}-1$ and then decrementing the counter by K for each point in the transform.

3.2.2 The Errata-Location Section

The errata-location section implements a modified version of the Berlekamp-Massey minimal length shift register synthesis algorithm to correct symbol errors and symbol erasures. First, this section uses the known erasure locations to calculate the erasure-locator polynomial. Then, the same hardware uses the error syndrome values to iteratively calculate the errata-locator polynomial. Finally, the errata-locator polynomial is used to generate the transform of the errata pattern which is subtracted from the transform of the received codeword in order to recover the original message.

The modified Berlekamp-Massey decoding algorithm was presented in Volume I and was reviewed in section II of this volume. In order to describe the hardware required to implement the errata-location section we define the decoding algorithm as a step-by-step procedure and then describe the implementation of each computation. This detailed decoding procedure is shown in Figure 7 and uses the notation defined in Table VII. The procedure of Figure 7 uses a forward transform for decoding (an inverse transform is defined for encoding), and the error syndrome symbols are defined as the first n-k symbols in the transform of the received sequence. (During encoding, the k information symbols are the k highest coefficients in the (n-1)th degree message polynomial.)

Given: R_i, $i = 0,1,\ldots,n-1$ Transform of received sequence

i_ℓ, $\ell = 0,1,\ldots,\nu-1$ Erasure locations

Initial Conditions: $\tau^{(-1)}(x) = \beta^{(-1)}(x) = b^{(-1)} = 1$

$L^{(-1)} = N = 0$

(1) If $N \geq n - k$ then: Go to (10)
else: $S_N = R_N$

(2) If $N = \nu$ then: Go to (6)
else: $d^{(N)} = \tau_N^{(N)}$

(3) $\tau^{(N)}(x) = \tau^{(N-1)}(x) - d^{(N)}b^{-(N-1)}x\beta^{(N-1)}(x)$

(4) $\beta^{(N)}(x) = \tau^{(N)}(x)$

$b^{(N)} = b^{(N-1)}$

$L^{(N)} = L^{(N-1)}$

(5) $N = N + 1$, Go to (1)

(6) $\tau^{(N)} = S_N + \sum_{j=1}^{L^{(N-1)}+\nu} \tau_j^{(N-1)} S_{N-j}$

(7) $\tau^{(N)}(x) = \tau^{(N-1)}(x) - d^{(N)}b^{-(N-1)}x\beta^{(N-1)}(x)$

If $2L^{(N-1)} < N + \nu$ then: $\beta^{(N)}(x) = x\beta^{(N-1)}(x)$

$L^{(N)} = L^{(N-1)}$

$b^{(N)} = b^{(N-1)}$

else: $\beta^{(N)}(x) = \tau^{(N-1)}(x)$

$L^{(N)} = N + \nu + 1 - L^{(N-1)}$

$b^{(N)} = d^{(N)}$

Go to (9)

(10) $\nu = N + k - n$

(11) $S_N = \sum_{i=1}^{L^{(N-1)}+\nu} \tau_i^{(N-1)} S_{N-i}$

(12) $M_\nu = R_N - S_N$

(13) If $N = n - 1$ then: (stop) Decoding is Complete
else: $\tau^{(N)}(x) = \tau^{(N-1)}(x)$

(14) $N = N + 1$

(15) Go to (11)

Figure 7: Decoding Algorithm

Table VII

Decoding Algorithm Variables

(Notation)

Volume I Notation	Volume II, Figure 7 Notation	Description
z	$x = z^{-1}$	Change in Variable
d^r	$d^{(N)}$	Discrepancy
d^m	$b^{(N)}$	*Previous Discrepancy*
$\sigma^{(r+1)}(z)$	$\Lambda^{(N)}(x)$	Present Connection Polynomial
$\sigma^{(m)}(z)$	$\beta^{(N)}(x)$	Previous Connection Polynomial
L	$L^{(N)}+\nu$	Length of Present Connection Polynomial
$\tilde{X}_i$	i_χ	Known Erasure Locations

The decoding algorithm shown in Figure 7 is required to operate for n iterations. During the first ν iterations, the known erasure locations are used to construct sequentially the erasure-locator polynomial. When the algorithm first branches to step (6), the polynomials $\Lambda^{(\nu-1)}(x)$ and $B^{(\nu-1)}(x)$ are both the erasure-locator polynomial. At the conclusion of the (n-k)th iteration, the polynomial $\Lambda^{(n-k-1)}(x)$ is the errata-locator polynomial. This polynomial is held constant for the remainder of the algorithm and the generated set $\{\hat{M}_0, \hat{M}_1, \ldots, \hat{M}_{k-1}\}$ is the recovered information.

A block diagram of the errata-location section is shown in Figure 8. The decision and control circuitry implied by the decoding algorithm are not shown in this figure but are implicit in the circuit operation. Both the algorithm presented in Figure 7 and the structure shown in Figure 8 are independent of the field of operation; the arithmetic operations specified in the algorithm and implemented in the block diagram are implicitly field-dependent.

Three major computational steps implement the decoding algorithm as shown in Figure 7:

- The first computation corresponds to step (6) and is the calculation of the present discrepancy, $d^{(N)}$.
- The second calculation corresponds to steps (3) and (8) and is the calculation of the present feedback connection polynomial, $\Lambda^{(N)}(x)$.
- The remaining calculation updates the previous feedback connection polynomial $B^{(N)}(x)$ to one of three values; $xB^{(N-1)}(x)$, $\Lambda^{(N-1)}(x)$, or $\Lambda^{(N)}(x)$.

The present discrepancy is always calculated prior to calculating the present feedback connection polynomial which in turn is calculated

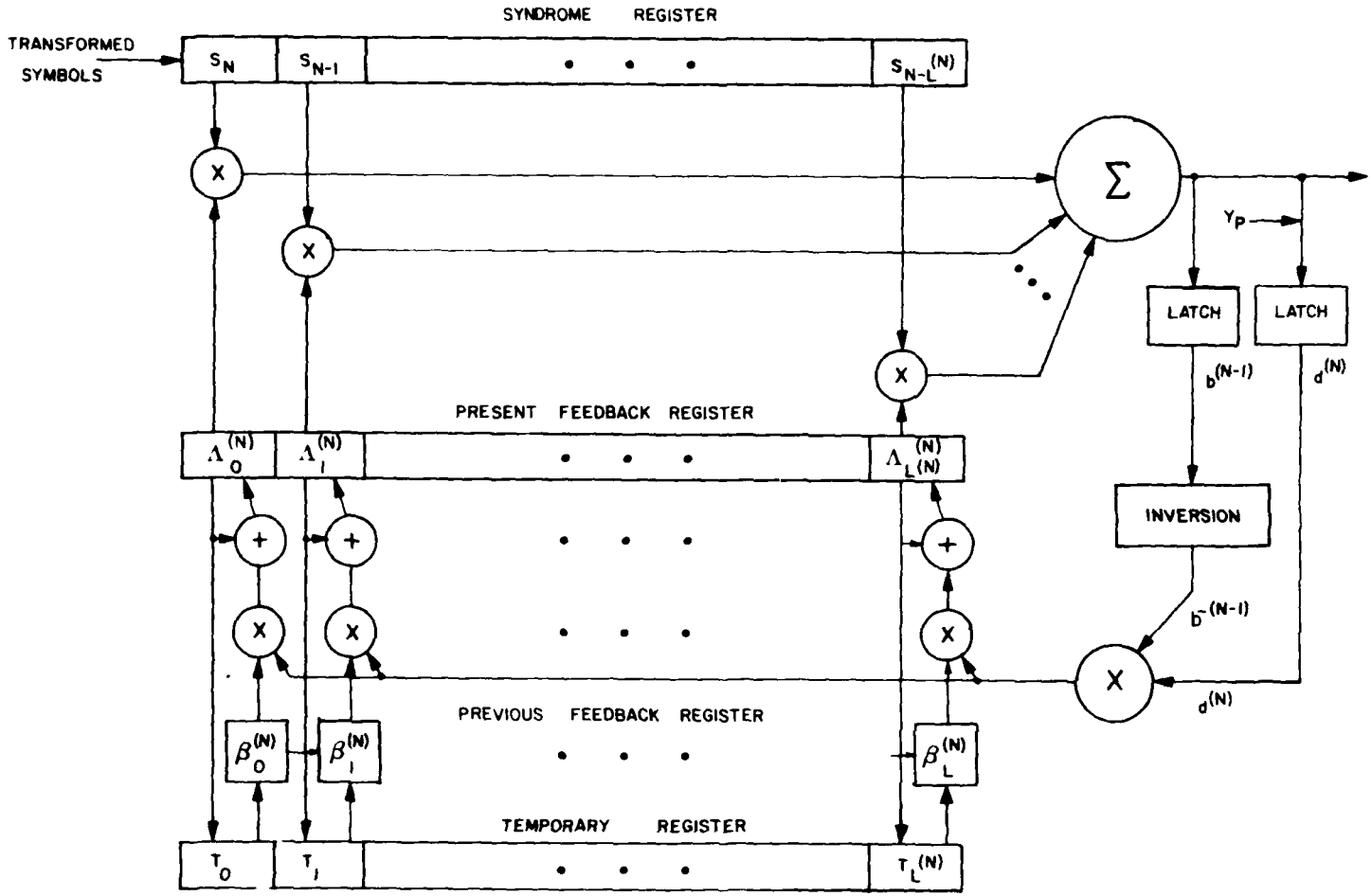


Figure 8. Errata Location Section

before updating the previous feedback connection polynomial. The hardware that implements these calculations is described in the remainder of this section.

3.2.2.1 Calculation of $d^{(N)}$. During each of the first n-k iterations of the decoding algorithm, the present discrepancy $d^{(N)}$ must be generated. For the first ν iterations, the known erasure locations, Y_p, are substituted for $d^{(N)}$. The erasure locations serve as inputs to the present discrepancy latch (see Figure 8). Once all erasure locations have been used as inputs, the present discrepancy is calculated as

$$d^{(N)} = S_N + \sum_{i=1}^{L^{(N-1)}+\nu} \Lambda_i^{(N-1)} S_{N-i} \qquad (3\text{-}12)$$

which corresponds to the convolution of the $L^{(N-1)} + \nu + 1$ most recent syndrome values $\{S_N, S_{N-1}, \ldots, S_{N-L^{(N-1)}-\nu}\}$ and the coefficients of the present feedback-connection polynomial $\Lambda^{(N-1)}(x)$. The portion of the structure shown in Figure 8 that is used to implement this correlation is repeated in Figure 9. In this figure, the $L^{(N-1)} + \nu + 1$ most recent syndrome values are held in the syndrome register, while the coefficients of $\Lambda^{(N-1)}(x)$ are held in the present feedback-polynomial register. Each of the syndrome values is multiplied with a corresponding coefficient of the feedback connection polynomial and the $L^{(N-1)} + \nu + 1$ product terms are summed to produce $d^{(N)}$. Each of the pairwise products between the syndrome register and the connection polynomial register is a multiplication in $GF(2^m)$, and $L^{(N-1)} + \nu + 1$ field-dependent multipliers are required for this direct implementation. The structure shown in Figure 9 illustrates the computational steps required to calculate $d^{(N)}$; the hardware is complex and may be simplified.

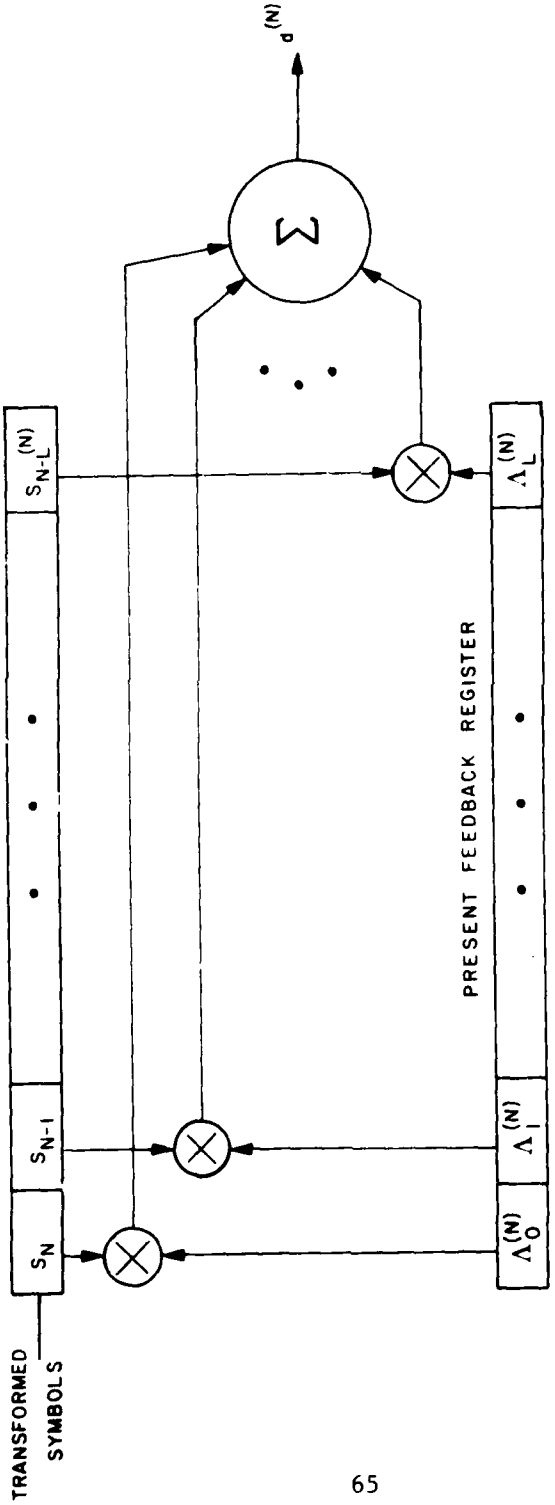


Figure 9. Present Discrepancy Calculator

The (255,k) encoder and decoder's present discrepancy calculator is shown in Figure 10. This structure implements sequential finite-field multiplication so that only one $GF(2^m)$ multiplier structure is required. In Figure 10, each syndrome value and polynomial coefficient is represented using our standard eight-bit symbol notation, (equation 3-9). The calculation of the present discrepancy, equation (3-12), can be expanded as

$$d^{(N)} = S_N + \sum_{i=1}^{L^{(N-1)}+\nu} \sum_{k=0}^{7} S_{N-i,7-k} \left(\sum_{\ell=0}^{7} \Lambda_{i,\ell}^{(N-1)} x^{\ell} \right) x^{7-k} \bmod p(x) \tag{3-13a}$$

$$= S_N + \sum_{k=0}^{7} \sum_{i=0}^{L^{(N-1)}+\nu} S_{N-i,7-k} \left(\sum_{\ell=0}^{7} \Lambda_{i,\ell}^{(N-1)} x^{\ell} \right) x^{7-k} \bmod p(x) \tag{3-13b}$$

where $p(x)$ is the primitive polynomial that generates $GF(2^m)$, $S_{N-i,k}$ is the k-th bit in the eight-bit representation of S_{N-i} and $\Lambda_{k,\ell}^{(N-1)}$ is the ℓ-th bit in the eight-bit representation of $\Lambda_k^{(N-1)}$. This equation is implemented directly in Figure 10 and this structure can be described in terms of its operation. During the first n-k iterations of the decoding algorithm, the syndrome values are used to calculate $d^{(N)}$ and switch 1 in Figure 10 is in position "1". During the N-th iteration ($N < n-k$), S_N is fed sequentially into the present discrepancy calculator. Simultaneously, each syndrome value already present in the syndrome register is shifted one stage to the right. Each syndrome value, S_j, is stored so that $S_{j,7}$ is the first bit shifted out. There are eight shifts required to shift each syndrome value and these shifts correspond to the summation over k in equation (3-13b). During each shift, a single bit of each syndrome value is multiplied modulo-2 with every bit from the corresponding pairwise product-polynomial coefficient $\Lambda_{i,\ell}^{(N-1)}$, ℓ=0, 1, ..., 7, and the eight partial products, $P_{i,0}$, $P_{i,1}$, ... $P_{i,7}$, from each of the

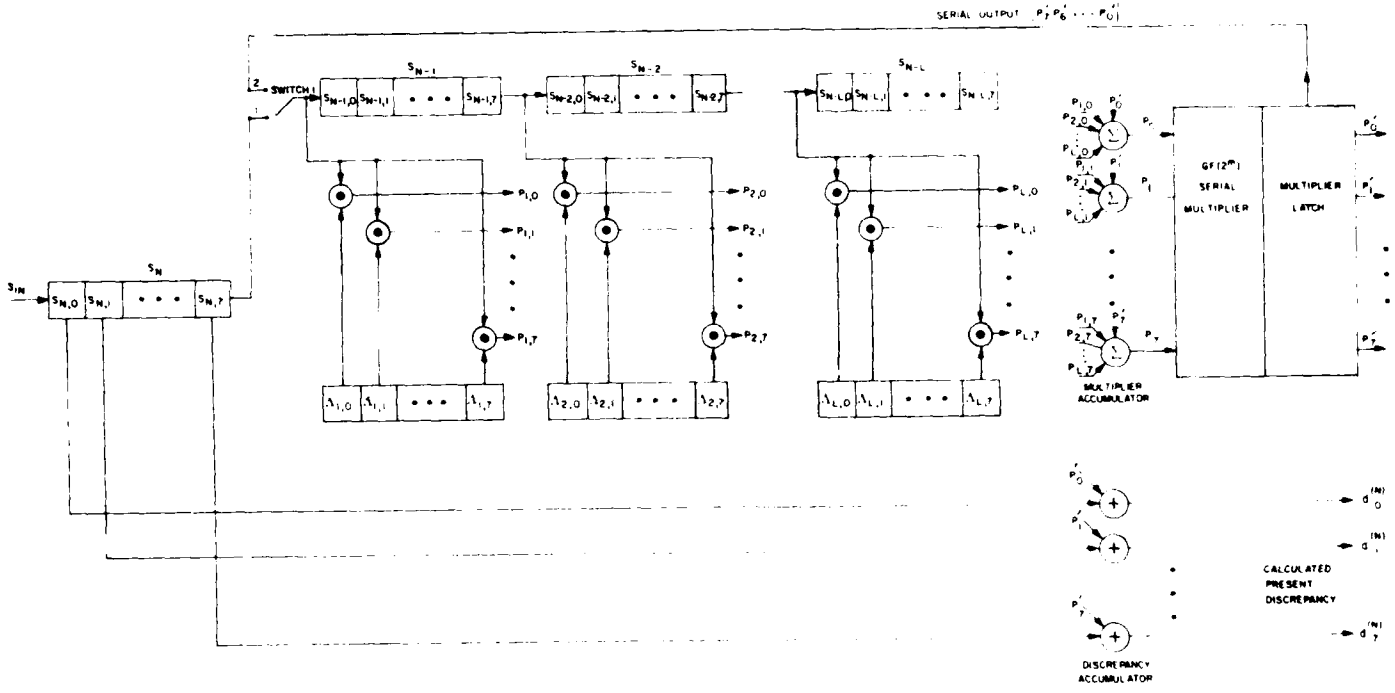


Figure 10. (255,k) Decoder's Present Discrepancy Calculator

$L^{(N-1)} + \nu$ stages are fed into the accumulator. For k=0, the summation

$$\sum_{i=1}^{L^{(N-1)}+\nu} S_{N-i,7} \left(\sum_{\ell=0}^{7} \Lambda_{i,\ell}^{(N-1)} x^{\ell} \right) \tag{3-14a}$$

$$= \sum_{\ell=0}^{7} \left(\sum_{i=0}^{L^{(N-1)}+\nu} S_{N-i,7} \Lambda_{i,\ell}^{(N-1)} \right) x^{\ell} \tag{3-14b}$$

is formed in the serial multiplier's accumulator. Equation (3-14b) represents a polynomial whose ℓth coefficient is given by

$$P_{\ell} = \sum_{i=1}^{L^{(N-1)}+\nu} S_{N-i,7} \Lambda_{i,\ell}^{(N-1)} \tag{3-15}$$

The eight coefficients $P_0, P_1, \ldots, P_7$ are fed into the $GF(2^m)$ serial multiplier where multiplication with x and polynomial reduction modulo p(x) occur. This product is stored in the multiplier's output latch. (A description of the $GF(2^m)$ serial multiplier is given in appendix A of this volume). During the k=1 shift, the summation

$$\sum_{i=0}^{L^{(N-1)}+\nu} S_{N-i,6} \left(\sum_{\ell=0}^{7} \Lambda_{i,\ell}^{(N-1)} \right) x^{\ell} \tag{3-16}$$

$$+ \sum_{i=1}^{L^{(N-1)}+\nu} S_{N-i,7} \left(\sum_{\ell=0}^{7} \Lambda_{i,\ell}^{(N-1)} x^{\ell} \right) x \bmod p(x)$$

is formed in the serial multiplier's accumulator. This term is fed to the $GF(2^m)$ serial multiplier, and the output is stored in the multiplier latch. This process continues and after seven shifts the $GF(2^m)$ serial multiplier latch contains the term

$$\sum_{k=0}^{6} \sum_{i=1}^{L^{(N-1)}+\nu} S_{N-i,7-k} \left(\sum_{\ell=0}^{7} \Lambda_{i,\ell}^{(N-1)} x^{\ell} \right) x^{7-k} \bmod p(x) \qquad (3\text{-}17)$$

During the k=7 shift, the contents of the multiplier latch, equation (3-17), are accumulated with

$$\sum_{i=1}^{L^{(N-1)}+\nu} S_{N-i,0} \left(\sum_{\ell=0}^{7} \Lambda_{i,\ell}^{(N-1)} x^{\ell} \right) \qquad (3\text{-}18)$$

and the sum is fed to the discrepancy accumulator, where S_N is added and the present discrepancy is formed. The architecture shown in Figure 10 requires only binary, GF(2), logic for each of the $L^{(N-1)}+\nu$ convolver stages. All field-dependent operations are implemented in the single $GF(2^m)$ serial multiplier.

3.2.2.2 Calculation of the Present Feedback Connection Polynomial, $\Lambda^{(N)}(x)$. During each of the first n-k iterations of the decoding algorithm, the present feedback connection polynomial $\Lambda^{(N)}(x)$ must be updated. This polynomial can be revised to one of two values,

$$\Lambda^{(N)}(x) = \Lambda^{(N-1)}(x) \qquad (3\text{-}19a)$$

or

$$\Lambda^{(N)}(x) = \Lambda^{(N-1)}(x) - d^{(N)}b^{-(N-1)}x\beta^{(N-1)}(x) \qquad (3\text{-}19b)$$

The portion of Figure 8 that updates $\Lambda^{(N)}(x)$ is repeated in Figure 11. This structure consists of the present connection polynomial register, the previous connection polynomial register, the present and past discrepancy latches, a field-element inversion circuit and a field-element multiplier.

The operation of the structure shown in Figure 11 can be described for both possible update conditions shown in equations (3-19). The first equation is trivial to implement. Prior to the N-th iteration, the polynomial $\Lambda^{(N-1)}(x)$ is stored in the present feedback polynomial register. If the conditions of the decoding algorithm are such that $\Lambda^{(N)}(x)$ is revised in accordance with equation (3-19a) then the contents of the present feedback polynomial register are not changed. The implementation of equation (3-19b) is more complex and it is best described by a three step procedure. First, the stored previous discrepancy $b^{(N-1)}$ is inverted and multiplied with the present discrepancy $d^{(N)}$. The inversion and multiplication are $GF(2^m)$ operations and they are implemented using the field-element inversion and $GF(2^m)$ multiplier shown in Figure 11. Next, the product $d^{(N)}b^{-(N-1)}$ is multiplied with $x\beta^{(N-1)}(x)$. The polynomial $\beta^{(N-1)}(x)$ is stored in the previous feedback polynomial register and $x\beta^{(N-1)}(x)$ is formed by shifting each coefficient of $\beta^{(N-1)}(x)$ one stage to the right. The product

$$d^{(N)}b^{-(N-1)}x\beta^{(N-1)}(x) \qquad (3\text{-}20)$$

is formed by multiplying the term $d^{(N)}b^{-(N-1)}$ with each coefficient of the shifted version of $\beta^{(N-1)}(x)$ using the corresponding $GF(2^m)$ multiplier. Finally equation (3-19b) requires that the polynomial formed in equation (3-20) be subtracted from the polynomial $\Lambda^{(N-1)}(x)$.

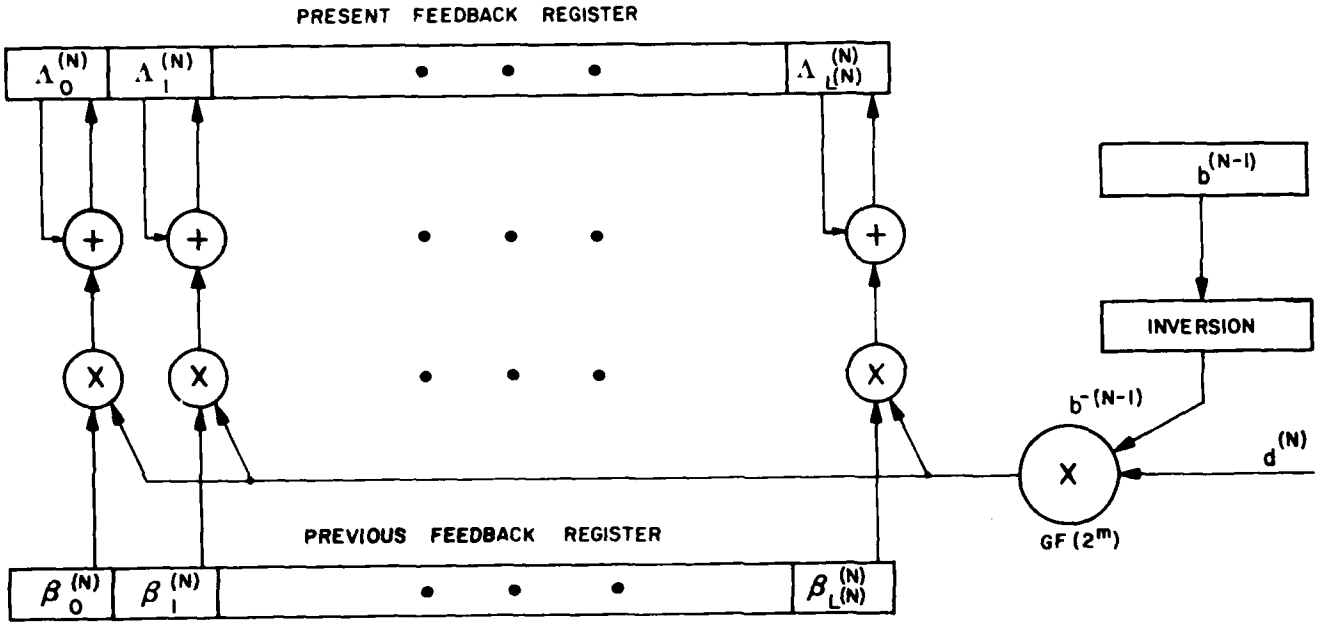


Figure 11. Present Feedback Connection Polynomial Calculator

Since the coefficients of both polynomials are from $GF(2^m)$, the arithmetic operation of subtraction is equivalent to modulo-two addition. Therefore, the desired results can be obtained as a coefficient-by-coefficient modulo-two addition of the two polynomials. The resulting polynomial $\Lambda^{(N)}(x)$ is then stored in the present feedback connection polynomial register.

The diagram of Figure 11 illustrates the computational steps required to update the present feedback connection polynomial. However, this structure requires $L^{(N-1)} + \nu + 2$ $GF(2^m)$ multipliers, making its hardware implementation complex. The expanded diagram shown in Figure 12 is functionally equivalent to the structure shown in Figure 11, but the structure of Figure 12 which uses sequential operation on symbol bits has been designed so that only two $GF(2^m)$ multipliers are used.

The first field-dependent structure calculates $d^{(N)}b^{-(N-1)}$. This structure uses an inversion-by-squared product algorithm to calculate the multiplicative inverse of $b^{(N-1)}$, and then the same structure calculates $d^{(N)}b^{-(N-1)}$. The heart of this circuit is a programmable $GF(2^m)$ array multiplier that is identical to the array multiplier designed for the transform section. This multiplier is combined with latches and field-element squaring circuitry to calculate $b^{-(N-1)}$. The same array multiplier is then used to calculate the product $d^{(N)}b^{-(N-1)}$. The field-element inversion algorithm and the details associated with the programmable array multiplier and the field-element squaring circuit are contained in appendix A of this volume.

The second field-dependent portion of Figure 12 is designed to simultaneously implement the product shown in equation (3-20) and the

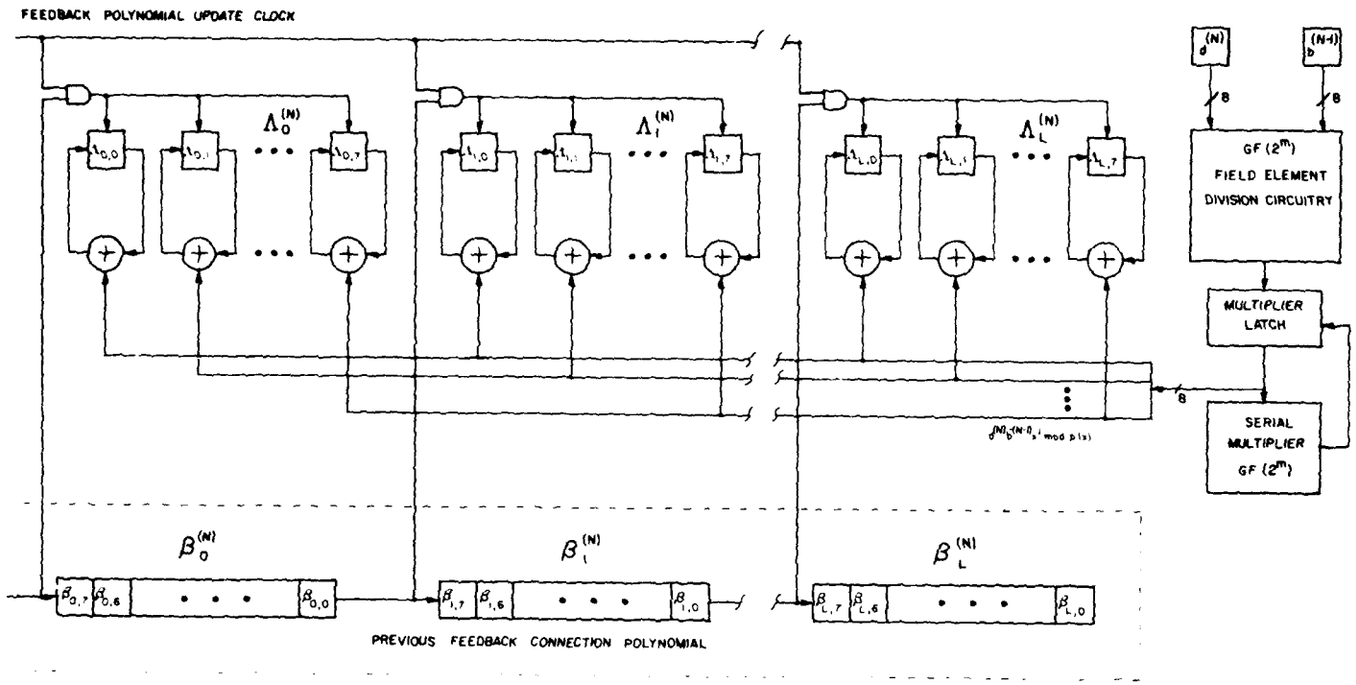


Figure 12. (255,k) Decoder's Present Feedback Connection Polynomial Calculator

coefficient-by-coefficient summation

$$\Lambda_i^{(N)} = \Lambda_i^{(N-1)} + \delta^{(N)}\beta_{i-1}^{(N-1)} \qquad (3\text{-}21)$$

where $\delta^{(N)} = d^{(N)}b^{-(N-1)}$. This equation can be expanded as

$$\Lambda_i^{(N)} = \Lambda_i^{(N-1)} + \sum_{j=0}^{7} \beta_{i-1,j}^{(N-1)} \left(\sum_{k=0}^{7} \delta_k^{(N)} x^k \right) x^j \bmod p(x) \qquad (3\text{-}22)$$

where $\beta_{i-1,j}^{(N-1)}$ is the j-th bit in our eight-bit representation (equation 3-9) of the (i-1)th coefficient from $\beta^{(N-1)}(x)$ and p(x) is the primitive polynomial that generates the field. The structure shown in Figure 12 implements equation (3-22) directly with the summation over j implemented sequentially in time. For each j the term

$$\delta^{(N)}x^j \bmod p(x) = \sum_{k=0}^{7} \delta_k^{(N)} x^{j+k} \bmod p(x) \qquad (3\text{-}23)$$

is multiplied with every coefficient of $\beta^{(N-1)}(x)$. The i-th stage of the present feedback connection polynomial register contains our eight-bit representation of $\Lambda_i^{(N-1)}$. For each j, a bit from the eight bit representation of $\beta_{i-1}^{(N-1)}$ is shifted one stage to the right within the previous feedback connection polynomial register. (These coefficients are stored so that the first bit shifted out is $\beta_{i-1,0}^{(N-1)}$.) Simultaneously, with each shift the j-th bit from $\beta_{i-1}^{(N-1)}$ is multiplied modulo-two with the value shown in equation (3-23) and the product is accumulated with $\Lambda_i^{(N-1)}$. This modulo-two multiplication and accumulation is implemented by using the bits $\beta_{i-1,j}^{(N-1)}$ as clocking gates for the single-bit accumulators that are the building blocks of

the present feedback connection polynomial (see Figure 12). After eight total shifts, the present feedback connection polynomial has been updated in accordance with equation (3-19b) and stored in the present feedback connection polynomial register. Also after eight shifts, the previous connection polynomial register contains $x\beta^{(N-1)}(x)$.

An example illustrates the operation of this structure (refer to Figure 12). For j=0, $\delta^{(N)}$ is held in the GF(2^m) serial multiplier latch. The contents of this latch are fed to every stage of the present feedback polynomial register. The contents of the i-th stage of the present feedback polynomial register is $\Lambda_i^{(N-1)}$. During the j=0 shift, the bit $\beta_{i-1,0}^{(N-1)}$ is shifted from the (i-1)th stage to the i-th stage of the previous feedback connection polynomial register. Simultaneously this bit, $\beta_{i-1,0}^{(N-1)}$, gates the clock for the i-th stage of the present feedback polynomial register. The summation

$$\Lambda_i^{(N-1)} + \delta^{(N)}\beta_{i-1,0}^{(N-1)} \tag{3-24}$$

is calculated and stored in place of $\Lambda_i^{(N-1)}$. During the j=1 shift, the contents of the GF(2^m) serial multiplier latch are fed to the multiplier. The product $\delta^{(N)}x \bmod p(x)$ is formed and stored in the multiplier latch. The bit $\beta_{i-1,1}^{(N-1)}$ is shifted in the previous feedback connection polynomial register and gated with the clock for the i-th stage of the present feedback connection polynomial register. The summation

$$\Lambda_i^{(N-1)} + \delta^{(N)}\beta_{i-1,0}^{(N-1)} + (\delta^{(N)}x \bmod p(x))\beta_{i-1,1}^{(N-1)} \tag{3-25}$$

is formed and stored in the i-th stage of the present feedback register. This process continues for eight total shifts and $\Lambda^{(N)}(x)$ is calculated in accordance with equation (3-19b).

3.2.2.3 Calculation of the Previous Feedback Connection Polynomial, $\beta^{(N)}(x)$. During each of the first n-k iterations of the decoding algorithm, the previous feedback connection polynomial $\beta^{(N)}(x)$ must be updated. The polynomial can be revised to one of three values as indicated in equation (3-26).

$$\beta^{(N)}(x) = x\,\beta^{(N-1)}(x) \tag{3-26a}$$

$$\beta^{(N)}(x) = \Lambda^{(N-1)}(x) \tag{3-26b}$$

$$\beta^{(N)}(x) = \Lambda^{(N)}(x) \tag{3-26c}$$

The conditions for determining the revised value of $\beta^{(N)}(x)$ depend upon the calculation of the present discrepancy, $d^{(N)}$, and the revision of the present feedback connection polynomial, $\Lambda^{(N)}(x)$. The procedure for revising $\beta^{(N)}(x)$ is carried out only after the other two calculations have been concluded. The hardware required for computing $\beta^{(N)}(x)$ consists of the two feedback connection polynomial registers and the temporary polynomial register (see Figure 8). The revision of $\beta^{(N)}(x)$ is closely related to the calculation of $\Lambda^{(N)}(x)$. During each iteration of the decoding algorithm, $\Lambda^{(N-1)}(x)$ is copied into the temporary polynomial register, and $d^{(N)}$ and $\Lambda^{(N)}(x)$ are then calculated. Temporary memory is required because the contents of the present connection polynomial register, $\Lambda^{(N-1)}(x)$, may be altered during the calculation of $\Lambda^{(N)}(x)$. After $\Lambda^{(N)}(x)$ has been calculated, the previous connection polynomial register contains $x\beta^{(N-1)}(x)$. If the conditions of the decoding algorithm are such that $\beta^{(N)}(x)$ is updated in accordance with equation (3-26a) then the revision is complete. If the conditions of the algorithm are such that equation (3-26b) is valid, then the contents of the temporary memory are transferred into the previous feedback connection polynomial register and $\beta^{(N)}(x)$ is equal to $\Lambda^{(N-1)}(x)$. Finally, if equation (3-26c) is to be implemented, $\Lambda^{(N)}(x)$ is transferred through the temporary memory into the previous feedback connection polynomial register.

3.2.2.4 Symbol Errata Correction. The decoding algorithm shown in Figure 7 requires n-k iterations to compute the errata-locator polynomial. If the total number of errors and erasures is within the bound of the code (equation 3-1), then the synthesized errata-locator polynomial is unique. The synthesized errata-location section will sequentially generate the transform of the errata pattern which is then subtracted from the transform of the received sequence to obtain the original message.

Errata correction and information recovery occur during the last k iterations of the decoding algorithm. Step (10) through (13) in the algorithm (see Figure 7) need to be implemented. The algorithm branches to step (10) after n-k iterations. At this time, the present feedback connection polynomial, $\Lambda^{(n-k-1)}(x)$, is the synthesized errata-locator polynomial. For $N \geq n-k$, step (11) of the decoding algorithm is

$$S_N = \sum_{i=1}^{L^{(N-1)}+\nu} \Lambda_i^{(N-1)} S_{N-i} \qquad (3\text{-}27)$$

Equation (3-27) defines the calculation required to generate the next symbol in the transform of the errata pattern. In step (12), this symbol is subtracted (added modulo-two) from the N-th symbol in the transform of the received sequence; the original information symbol is recovered. During the last k iterations of the decoding algorithm, the present feedback connection polynomial, $\Lambda^{(N)}(x)$, remains unchanged.

The hardware that implements information recovery in the decoding algorithm is contained within the present discrepancy calculator as seen in Figure 10. The single switch controls the operation of the hardware. During the first n-k iterations of the algorithm, switch 1 is in position "1", and the syndrome values are fed into the syndrome

register. During the last k iterations of the algorithm, switch 1 is in position "2" and the input into the syndrome register is S_N in accordance with equation (3-27).

During the N-th iteration, (N>n-k), S_N is calculated in the serial $GF(2^m)$ multiplier's accumulator (see Figure 10). This calculation requires eight shifts and is implemented identically as the calculation of $d^{(N)}$ (see section 3.2.2.1). Also during the N-th iteration, the N-th symbol in the transform of the received pattern, R_N, is entered into the present discrepancy calculator. On the eighth shift of the N-th iteration, S_N (as calculated in equation (3-27)) is added modulo-two to R_N in the $d^{(N)}$ accumulator. This accumulator implements

$$R_N + S_N = \hat{M}_\ell \qquad (3\text{-}28)$$

and an original information symbol is recovered. This process is repeated for k cycles; the entire original message is recovered.

3.3 Operational Characteristics

The Reed-Solomon (255,k) encoder and decoder is designed to operate continuously in a serial input data mode. The processing time required for both encoding and decoding can be described in terms of the operation of the transform section and the errata-location section. To facilitate this description it is advantageous to define a machine cycle as the maximum time required for the encoder and decoder to complete one cycle of the iterative decoding algorithm (see Figure 7). Figure 13 shows the timing requirements associated with the operation of the transform and errata-location sections.

The sequential calculation of an n-point transform requires 2n machine cycles. During each of the first n machine cycles, a symbol

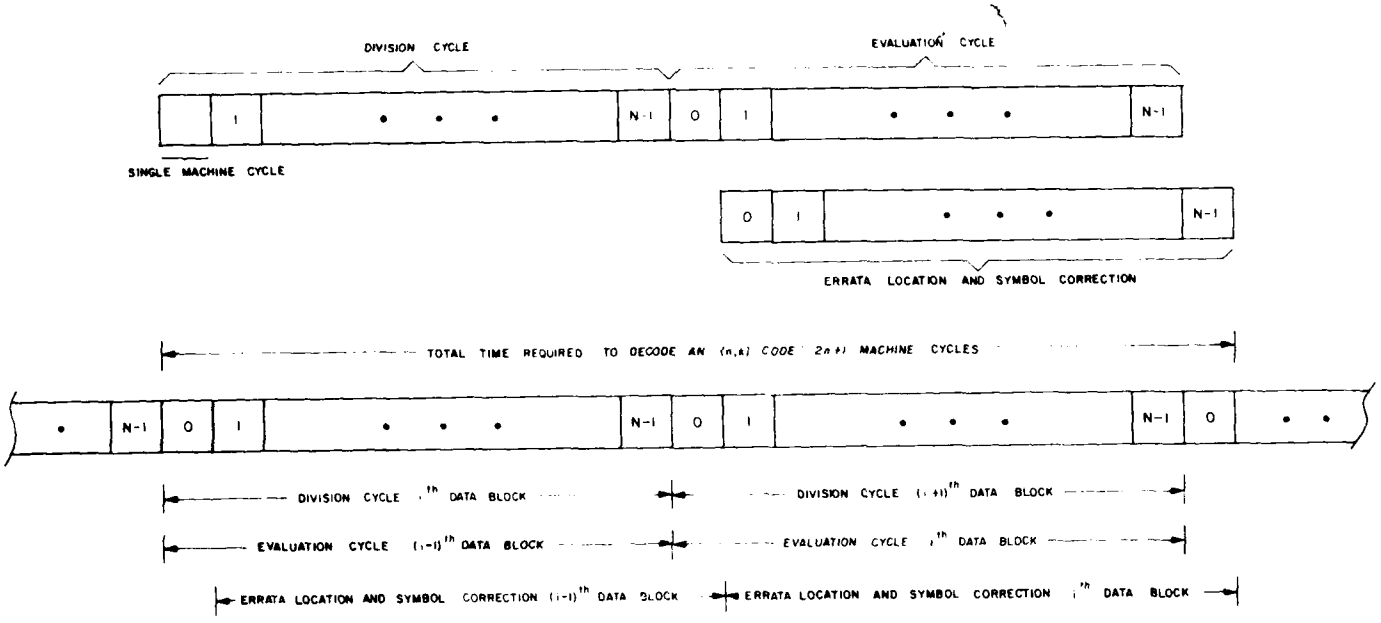


Figure 13. Timing Requirements for (255, k) Encoder and Decoder

from the sequence to be transformed is fed into the transform section's polynomial residue calculator. At the conclusion of n machine cycles, all n symbols of the sequence will have been used as inputs and the transform algorithm's polynomial division will be complete. At this time, the polynomial residues that have been calculated are transferred into a temporary memory so that they are available for further processing. The second n cycles define the evaluation period. During each of these machine cycles, a residue polynomial is selected from the residue calculator's temporary memory and evaluated to produce a single transform value. The evaluation process is calculated within the transform section's residue evaluator, and the i-th transformed symbol becomes available at the conclusion of the i-th machine cycle of the evaluation period. The transform section is a pipeline in which two adjacent blocks of n symbols are in process at all times.

The errata-location and symbol correction sections also operate with sequential symbols. During decoding, the errata-locator uses the first ν transformed symbols to calculate the erasure-locator polynomial. Then the errata-locator uses the next $n-k-\nu$ symbols to synthesize the errata-locator polynomial. The symbol correction circuitry uses the synthesized polynomial to correct the remaining k transformed symbols. Polynomial synthesis and symbol correction require n machine cycles. The sequential symbols required as input to the errata-locator are available at the completion of each machine cycle in the transform section's evaluation period. The n machine cycles associated with the operation of the errata-location section are offset one machine cycle from the n cycles that constitute the transform's evaluation period. The total time required to decode an (n,k) code requires $2n + 1$ machine cycles. In a continuous data mode, the transform section does not wait until it processes one block of data before it starts on the next one so, after an initial delay of $n + 1$ machine cycles, a block of decoded symbols becomes available every n machine cycles.

A single machine cycle is defined as the total number of clocking cycles required to implement the computational steps in a single iteration of the decoding algorithm. Figure 14 shows the relationship between a clock cycle and a machine cycle. The timing required to calculate the intermediate steps in the decoding algorithm is also shown in this figure. Finite-field multiplication is implemented within a single clock cycle. Using the programmable array multiplier structure described in appendix A, we can easily obtain multiplication rates of less than 100 nanoseconds using standard Schottky TTL logic. A VLSI implementation of the array multiplier can probably achieve 50 nanosecond multiplication times, corresponding to a clocking rate of 20 MHz. Twenty clock cycles are required to represent one machine cycle. The (255,k) Reed-Solomon decoder can completely decode an (n,k) code in (2n + 1) microseconds. With continuous operation a completely decoded block from an (n,k) code would be available every n microseconds. For example, the (255,k) Reed-Solomon decoder can decode a codeword from a (31,15) code in 63 microseconds using a projected 20 MHz clock. In a continuous mode, a decoded codeword would be available every 31 microseconds.

3.4 Hardware Complexity

The transform section and the errata-location section each could be fabricated as a single VLSI device. Much of the circuitry required for the transformer and errata-locator is highly repetitive, and both sections share functional circuits that can be designed once and then repeated.

Most of the circuitry required for the transformer is devoted to the implementation of the polynomial residue calculator. This structure, consisting of 35 divider circuits, can be designed using a macrocell with one bit of shift register, one bit of temporary memory, and eight elementary logic gates (see Figure 15). The macrocell represents a single programmable stage from a BFSR, and the tem-

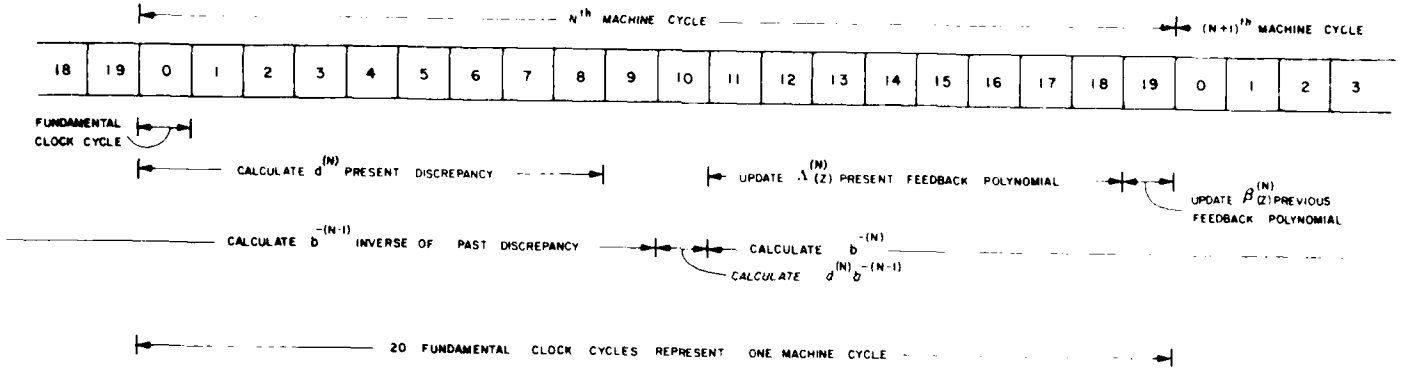


Figure 14. Errata Locator Timing, Definition of a Machine Cycle

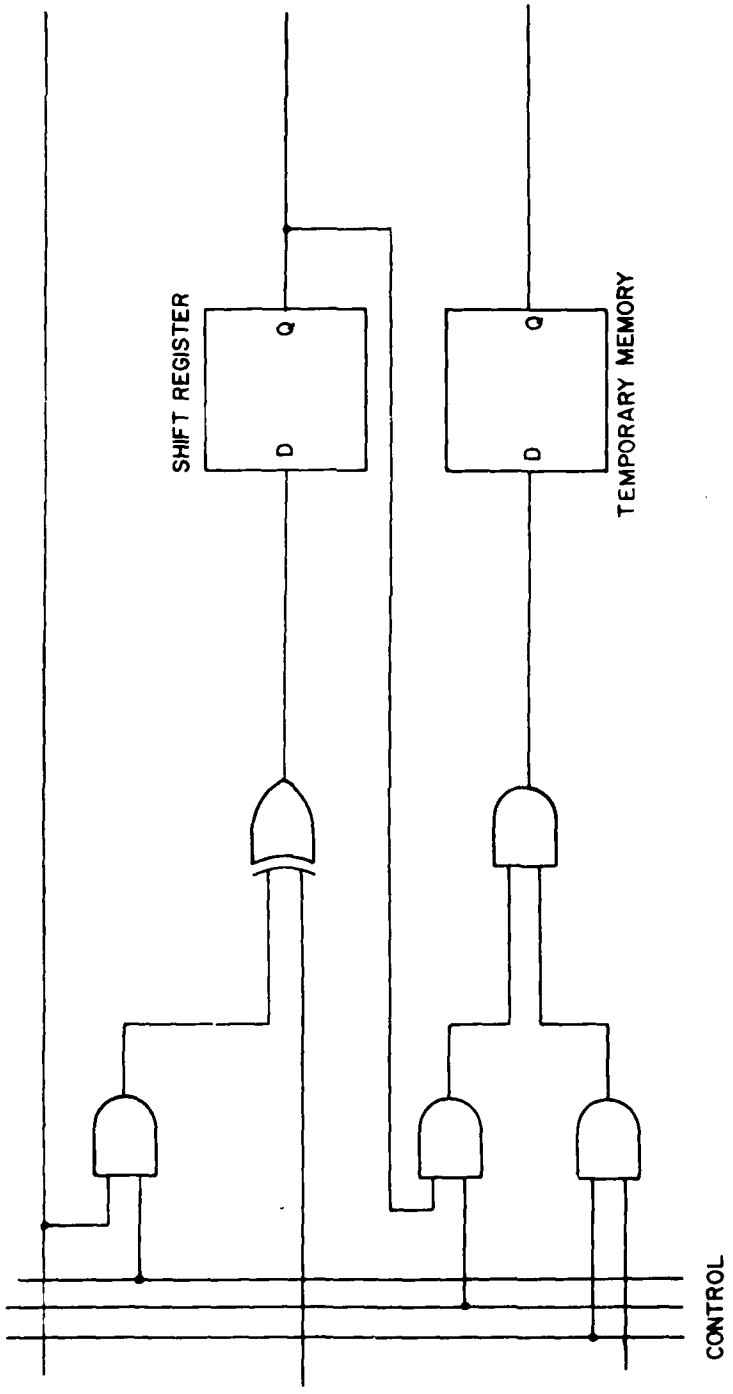


Figure 15. Divider Circuit Macrocell

porary memory required to operate the transformer in a pipeline fashion. Approximately 2.4k of these macrocells are required to implement the residue calculator. An additional 1k elementary logic gates are required to select the outputs of the divider circuits and to provide programmability for the different feedback connection polynomials. Due to the large number of shift register stages required to implement this section, the gate complexity will be heavily dependent on device technology. However, the design should be obtainable using current NMOS or other mature technologies.

The transformer's polynomial residue evaluator can be implemented with fewer than 1k logic gates. The accumulator portion of this section requires fewer than 100 gates, and the reconfigurable multiplier has been designed to be implemented with no more than 900 gates. All of the preceding complexity estimates are based upon a direct implementation with two-input NAND or NOR gates.

The transform section's arithmetic controller could be fabricated on the same integrated circuit as the residue calculator and the residue evaluator. Alternatively, the controller also could be implemented easily on a separate MSI chip containing a modest amount of programmable read-only memory [10]. A custom or semi-custom LSI implementation of the entire transformer would require several integrated circuits.

The architecture associated with the implementation of the decoding algorithm is shown in Figure 16. The hardware resembles an adaptive transversal filter. Reconfigurability for different code parameters is accomplished by separating the binary-extension field operations from other binary operations. As a result, most of the errata-location section is configured as a binary transversal filter (or convolver), and the remaining portion is reconfigurable to accommodate the necessary field-dependent operations.

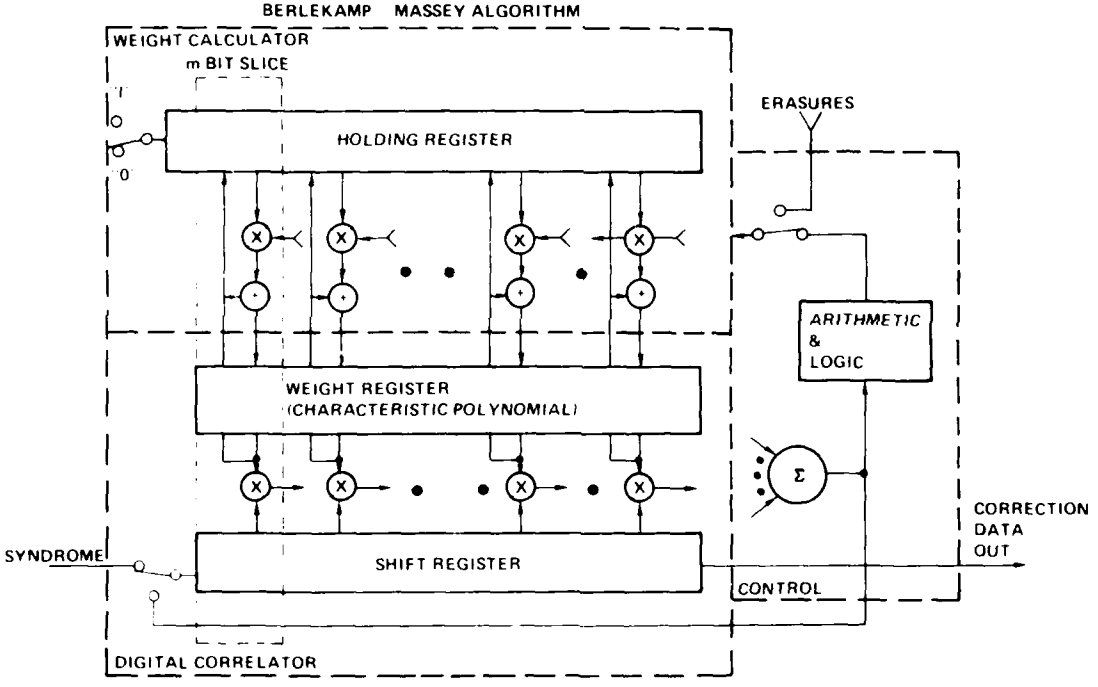


Figure 16. Errata Location Section Architecture

Most of the circuitry used for the errata-location section is dedicated to implementing the binary stages of the transversal filter. This circuitry is highly repetitive and benefits from the modularity and common busing structures inherent in VLSI architectures. The binary filter consists of 128 identical slices of hardware, favoring macrocell design. Each of the 128 slices consists of 32 bits of shift register and approximately 75 additional logic gates. Figure 17 is a logic diagram of a single slice of the 128 stage filter. Within each slice, a cell can be identified that consists of four bits of shift register and approximately eight logic gates. This cellular design can be repeated to implement the binary transversal filter. The entire 128-stage filter consists of approximately 4k bits of shift register and 10k bits of additional logic gates.

The field-dependent portion of the errata-locator consists of field-element inversion circuitry, a present discrepancy calculator, two reconfigurable $GF(2^m)$ serial multipliers, and necessary control logic. The most complex component of these structures is the field-element inversion circuitry. The heart of this structure is a programmable $GF(2^m)$ array multiplier that is identical to the structure required in the transformer's polynomial residue evaluator. The entire field-dependent portion, excluding control, consists of less than 2k logic gates. Again, control logic could be implemented in the same integrated circuit, or a separate device could be designed.

Both the transformer and the errata-locator have hardware complexities that suggest fabrication as single VLSI devices. Table VIII summarizes the approximate hardware complexity and features associated with the transformer and errata-locator. Each of the sections could easily be implemented as a set of LSI devices where many of the devices would be identical.

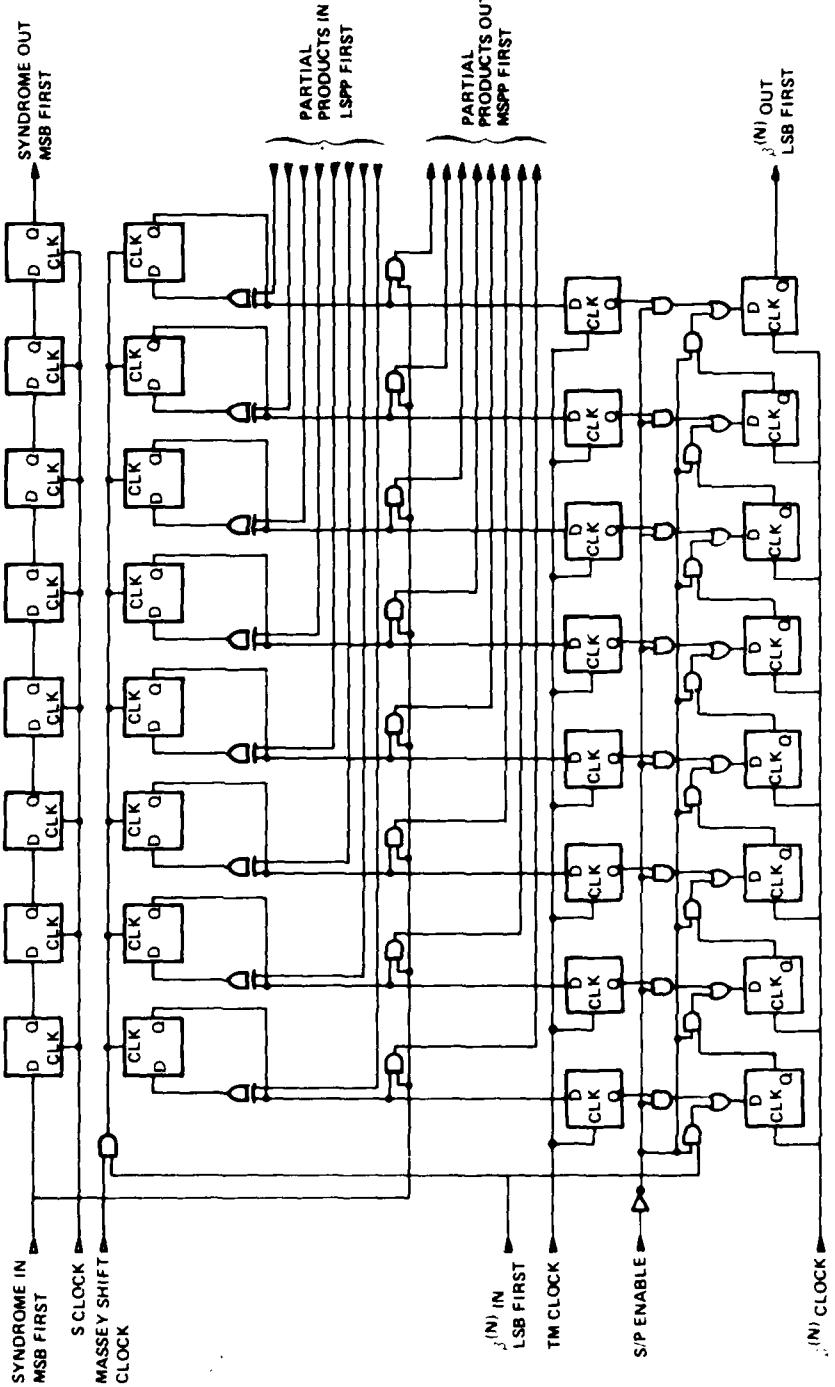


Figure 17. 8-Bit Symbol Correction Slice

Table VIII

(255,k) Encoder and Decoder Hardware Complexity

Function	Architecture	Complexity
Transformer	Programmable Over $GF(2^m)$ m = 4,5,6,7,8 Repetitive Structure Accommodate 588 codes	4.5k bits of shift register 13k gates
Errata Locator	Programmable over $GF(2^m)$ m= 4,5,6,7,8 Correct $2t + s \leq 128$ 128 identical slices of hardware	4.5k bits of shift register 15K gates

SECTION IV

A (51,k) REED-SOLOMON TRANSFORM ENCODER AND DECODER TTL BREADBOARD

A TTL breadboard that encodes and decodes a large number of Reed-Solomon symbol error-correction codes was designed and fabricated. This breadboard implements the transform encoding and decoding algorithms described in section III of this volume. The code of longest block length that can be processed by the breadboard is a 51-symbol code with each symbol represented by eight bits. The (51,k) Reed-Solomon transform encoder and decoder breadboard is shown in Figure 18.

The major difference between the TTL implementation and the design proposed for future VLSI implementation is size. The breadboard contains only eight polynomial divider circuits which process eight-bit symbols and it cannot calculate all of the n-point transforms that can be processed by the (255,k) encoder and decoder. The breadboard's transform section does not contain the additional temporary memory that allows pipeline operation. The breadboard's errata-location section is not as large as the (255,k) decoder's errata-locator; consequently the breadboard cannot correct as many combinations of errors and erasures as can be processed by the (255,k) decoder. The codes that can be processed by the TTL breadboard are shown in Table IX. Although the breadboard cannot encode or decode all of the codes processed by the (255,k) encoder and decoder, it can accommodate a large subset of them. It therefore serves as a proof-of-concept verification of the (255,k) encoder and decoder.

4.1 Transform Section

The (51,k) encoder and decoder's transform section is contained on the five wire-wrap logic boards shown in the lower right corner of Figure 18. This transformer implements the fast polynomial evaluation algorithm described in section 3.2.1. The breadboard's trans-

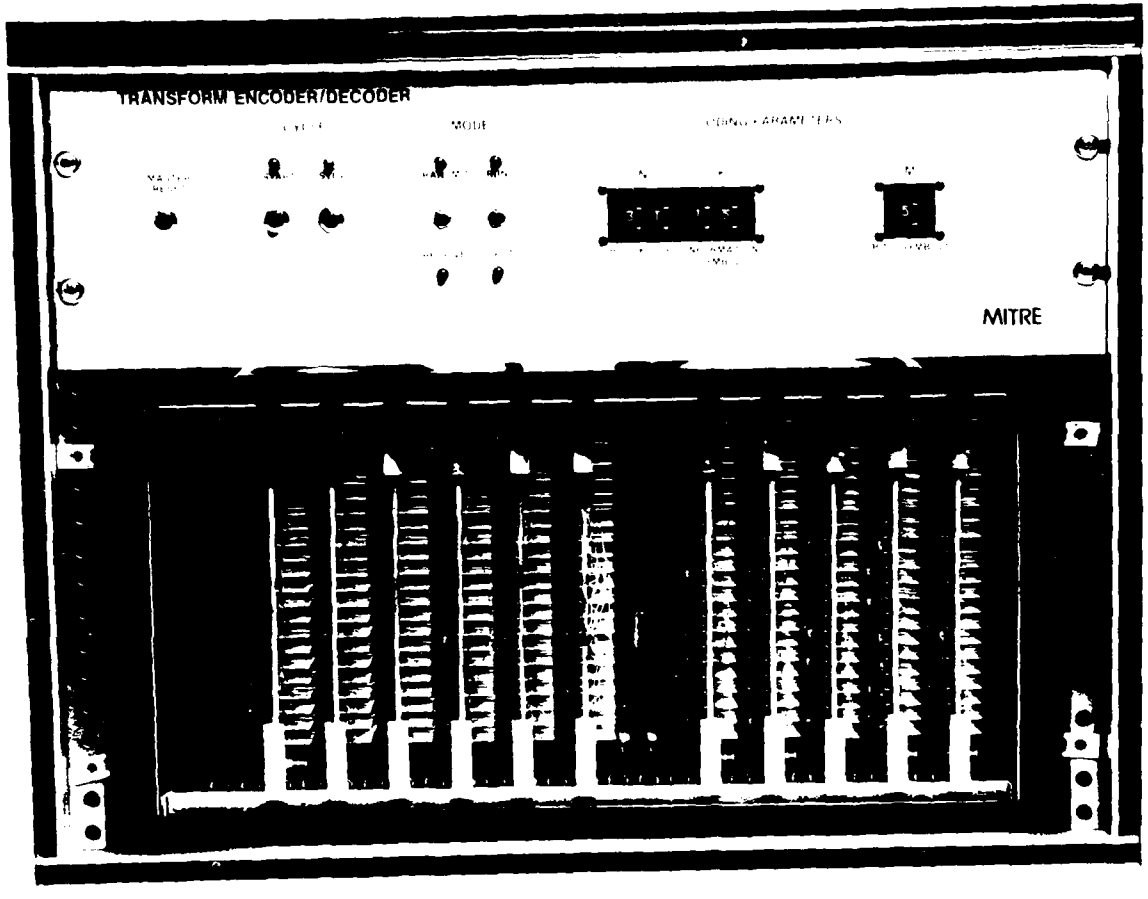


Figure 18. (51, k) TTL Breadboard

Table IX

(51,k) Breadboard Coding Capabilities

MAXIMUM BLOCK LENGTH (SYMBOLS)	BITS/SYMBOL M	(n, k) Codes $n - k \leq 16$	
51	8	(51, k) (17, k) (15, k) (5, k) (3, k)	55 TOTAL
21	6	(21, k) (9, k) (7, k)	32 TOTAL
31	5	(31, k)	16 TOTAL
15	4	(15, k) (5, k) (3, k)	23 TOTAL

former consists of a polynomial residue calculator, a polynomial residue evaluator, and an arithmetic controller.

The residue evaluator and arithmetic controller were designed identically to those structures described earlier in section 3.2.1.2 and section 3.2.1.3. The breadboard's residue calculator is a scaled version of the (255,k) encoder and decoder's polynomial residue calculator. The polynomials that can be used for division and the transforms that can be processed by the breadboard are shown in Table X. This table lists the lengths of the transforms, the fields in which the transforms are defined, the kernels of the transforms, and the minimal polynomials required for the fast polynomial evaluation algorithm.

4.1.1 Polynomial Residue Calculator

The breadboard's polynomial residue calculator consists of eight divider circuits. Each circuit consists of eight identical BFSRs that can be reconfigured for division using the divisor polynomials shown in Table X. The residue calculator is designed to operate with our eight-bit symbol representation (equation 3-9); for operation in $GF(2^m)$, where $m < 8$; - m binary shift registers are unused. The eight divider circuits were designed using the hardware reduction techniques described in section 3.2.1.1. The resulting implementation contains only three programmable feedback taps. The division capabilities of the divider circuits are shown in Table XI.

To compensate for the lack of memory required for pipeline operation, each of the breadboard's eight divider circuits operates in two modes. During the transform's division cycle all divider circuits are configured to divide by the minimal polynomials associated with the desired transform. After division is complete, each shift register is reconfigured so that its feedback taps are deactivated and each shift register's output is fed back to its input. Each divider

Table X

Transform Capabilities of the (51,k) Breadboard

Transform Size N	Bits Per Symbol	Kernel of Transform	Required Minimal Polynomial Divisors
51	8	α^5	$m_0(z)$, $m_5(z)$, $m_{15}(z)$, $m_{25}(z)$, $m_{45}(z)$, $m_{55}(z)$, $m_{85}(z)$, $m_{95}(z)$
17	8	α^{15}	$m_0(z)$, $m_{15}(z)$, $m_{45}(z)$,
15	8	α^{17}	$m_0(z)$, $m_{17}(z)$, $m_{51}(z)$, $m_{85}(z)$, $m_{119}(z)$
5	8	α^{51}	$m_0(z)$, $m_{51}(z)$
3	8	α^{85}	$m_0(z)$, $m_{85}(z)$
21	6	α^3	$m_0(z)$, $m_3(z)$, $m_9(z)$, $m_{15}(z)$, $m_{21}(z)$, $m_{27}(z)$
9	6	α^9	$m_0(z)$, $m_7(z)$, $m_{21}(z)$
7	6	α^7	$m_0(z)$, $m_9(z)$, $m_{27}(z)$
31	5	α^0	$m_0(z)$, $m_1(z)$, $m_3(z)$, $m_5(z)$, $m_7(z)$, $m_{11}(z)$, $m_{15}(z)$
15	4	α^0	$m_0(z)$, $m_1(z)$, $m_3(z)$, $m_5(z)$, $m_7(z)$
5	4	α^3	$m_0(z)$, $m_3(z)$
3	4	α^5	$m_0(z)$, $m_5(z)$

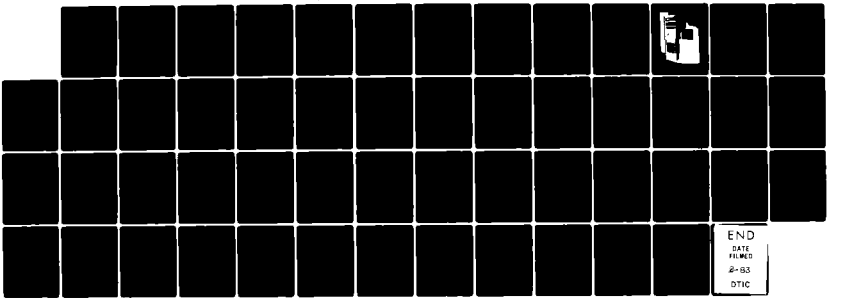
Table XI

Programmability of the (51,k)
Transformer's Divider Circuit

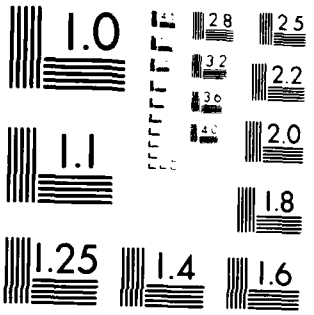
Divider Circuit	Divisor Polynomial	Field of Operation
1	$M_{85}(z) = 1 + z + z^2$	$GF(2^8)$
	$M_{15}(z) = 1 + z^2 + z^4 + z^5 + z^6$	$GF(2^6)$
	$M_1(z) = 1 + z^2 + z^5$	$GF(2^5)$
	$M_5(z) = 1 + z + z^2$ $(m_{85}(z))$	$GF(2^4)$, $(GF(2^8))$
2	$M_{95}(z) = 1 + z + z^2 + z^3 + z^4 + z^7 + z^8$	$GF(2^8)$
	$M_0(z) = 1 + z$	$GF(2^6)$
	$M_7(z) = 1 + z + z^2 + z^3 + z^5$	$GF(2^5)$
	$M_3(z) = 1 + z + z^2 + z^3 + z^4$ $(M_{51}(z))$	$GF(2^4)$, $(GF(2^8))$
3	$M_5(z) = 1 + z + z^4 + z^5 + z^6 + z^7 + z^8$	$GF(2^8)$
	$M_{27}(z) = 1 + z + z^3$	$GF(2^6)$
	$M_0(z) = 1 + z$	$GF(2^5)$
	$M_1(z) = 1 + z + z^4$ $(M_{15}(z))$	$GF(2^4)$, $(GF(2^8))$
4	$M_{15}(z) = 1 + z + z^2 + z^4 + z^6 + z^7 + z^8$	$GF(2^8)$
	$M_3(z) = 1 + z + z^2 + z^4 + z^6$	$GF(2^6)$
	$M_5(z) = 1 + z + z^2 + z^4 + z^5$	$GF(2^5)$
	$M_0(z) = 1 + z$	$GF(2^4)$
5	$M_{45}(z) = 1 + z^3 + z^4 + z^5 + z^8$	$GF(2^8)$
	$M_9(z) = 1 + z^2 + z^3$	$GF(2^6)$
	$M_3(z) = 1 + z^2 + z^3 + z^4 + z^5$	$GF(2^5)$
6	$M_{25}(z) = 1 + z + z^3 + z^4 + z^8$	$GF(2^8)$
	$M_{21}(z) = 1 + z + z^2$	$GF(2^6)$
	$M_{11}(z) = 1 + z + z^3 + z^4 + z^5$	$GF(2^5)$
7	$M_0(z) = 1 + z$	$GF(2^8)$
	$M_7(z) = 1 + z^3 + z^5$	$GF(2^6)$
	$M_{15}(z) = 1 + z^3 + z^5$	$GF(2^5)$
	$M_7(z) = 1 + z^3 + z^4$ $(M_{119}(z))$	$GF(2^4)$, $(GF(2^8))$
8	$M_{55}(z) = 1 + z^4 + z^5 + z^7 + z^8$	$GF(2^8)$

AD-A123 977 TRANSFORM DECODING OF REED-SOLOMON CODES VOLUME II LOGICAL DESIGN AND IMP..(U) MITRE CORP BEDFORD MA B L JOHNSON ET AL. NOV 82 MTR-8278-VOL-2

UNCLASSIFIED ESD-TR-82-403-VOL-2 F19628-82-C-0001 F/G 9/4 NL



END
DATE FILMED
2-83
DTIC



MICROCOPY RESOLUTION TEST CHART
NATIONAL BUREAU OF STANDARDS-1963-A

circuit becomes a recirculating shift register that contains the calculated residue polynomial. This residue can be read out of the recirculating memory to calculate a transform point; the residue is simultaneously restored for further processing. The ability to store the residue polynomials in the calculation hardware demonstrates the breadboard's hardware efficiency. However, the divider circuits cannot process new information while they are being used as recirculating memories. The duty cycle of the breadboard's transform section is one-half the duty cycle of the (255,k) transformer, and throughput is reduced proportionally.

The polynomial residue calculator is implemented with four of the five wire-wrap logic boards located in the lower right corner of the breadboard's card cage (Figure 18). The four boards are identical and each contains two identical slices of hardware. Each slice implements the eight reconfigurable BFSRs shown in Table XI. A logic-level diagram of a single slice of the residue calculator is shown in Figure 19. In this figure, the four-to-one multiplexer associated with each shift register selects the register's output tap which defines the shift register's divisor polynomial. The AND-gate located at the output of this multiplexer controls the BFSR's mode of operation. An activated AND-gate indicates polynomial division; a deactivated AND-gate indicates recirculating data in the BFSR. The final eight-to-one multiplexer is used to select the residue polynomials that are required to complete the polynomial evaluation algorithm.

4.1.2 Polynomial Residue Evaluator

The breadboard's polynomial residue evaluator is implemented using the fifth wire-wrap logic board shown in the lower right corner of Figure 18. The evaluator consists of an eight-bit modulo-two accumulator and a programmable GF(2^m) array multiplier. A detailed block diagram of the evaluator is shown in Figure 20. The evaluator implements a continued product expansion for polynomial residue

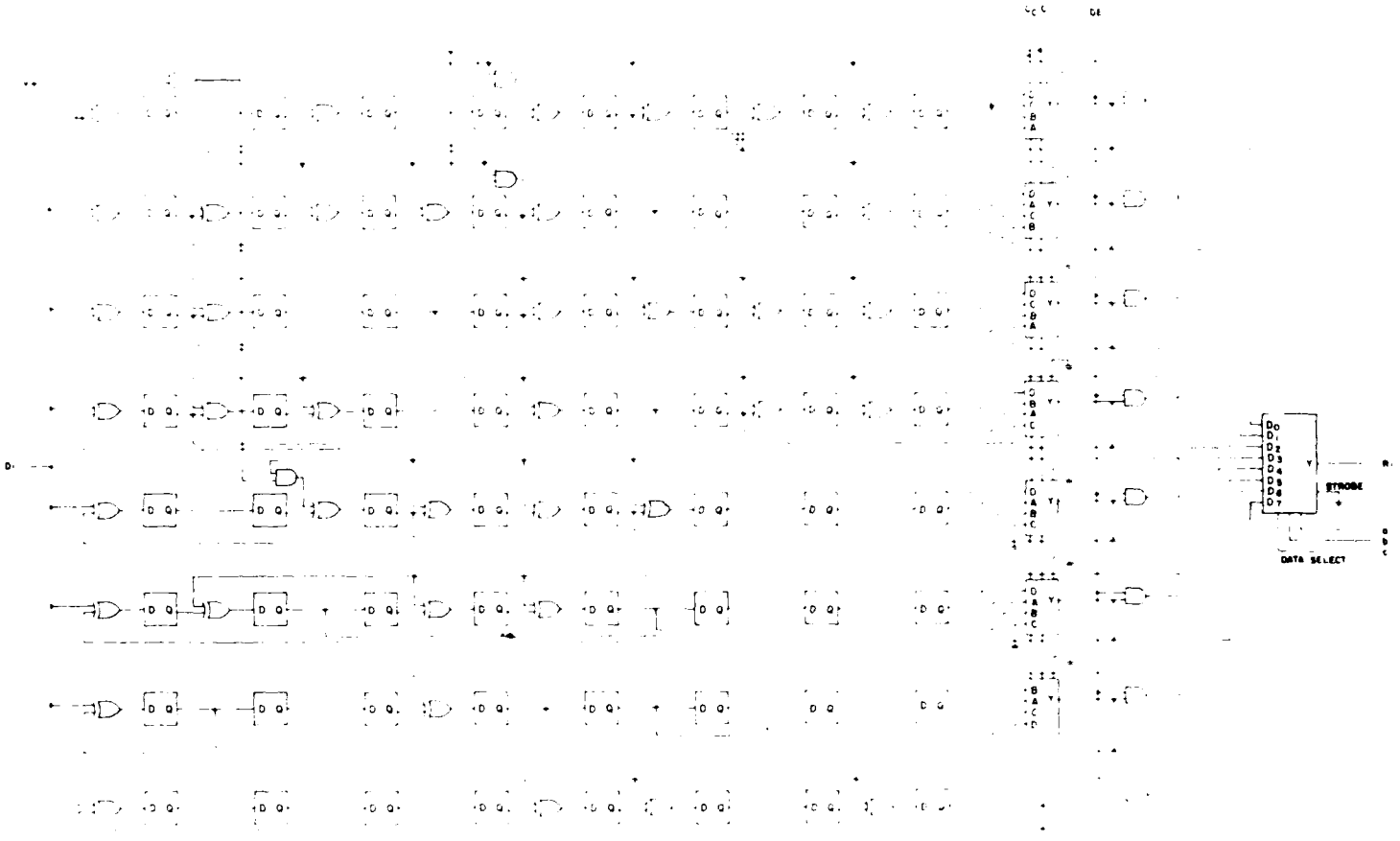


Figure 19. A Single Bit Slice of the (51, k) Transformer's Polynomial Divider Circuits

evaluation (equation 3-11). Design and operation of polynomial residue evaluators are identical in the breadboard and the (255,k) transformer.

The critical component of the breadboard's polynomial residue evaluator is the programmable GF(2^m) array multiplier. (A description of this multiplier is given in appendix A of this volume). This multiplier consists of a pairwise product array, an accumulator array, and programmable field-reduction circuitry (Figure 20). The pairwise product array operates with two 8-bit inputs and forms 64 pairwise modulo-2 products. This array is implemented as 64 2-input AND gates (see Figure 21). The accumulator array operates on the 64 pairwise products and forms 15 partial sums (see appendix A). The accumulator is implemented as 15 Exclusive-OR trees using 2-input Exclusive-OR gates. The programmable field reduction circuitry operates on the 15 partial sums and calculates the 8-bit representation of the desired product. This circuit is implemented as eight Exclusive-OR trees, whose inputs are programmed in accordance with the field reduction equations presented in appendix A.

4.1.3 Arithmetic Controller

The breadboard's arithmetic controller is located behind the front panel controls shown in Figure 18. The controller consists of a programmable up-down counter, a transform kernel-generating circuit and preprogrammed memory as shown in Figure 6. The arithmetic controller is implemented in discrete combinational logic and memory.

4.2 Errata-Location Section

The breadboard's errata-location section is implemented using the six wire-wrap logic boards shown in the lower left corner of Figure 18. The breadboard's errata-locator contains 16 symbol error-correction slices while the (255,k) decoder's errata locator contains

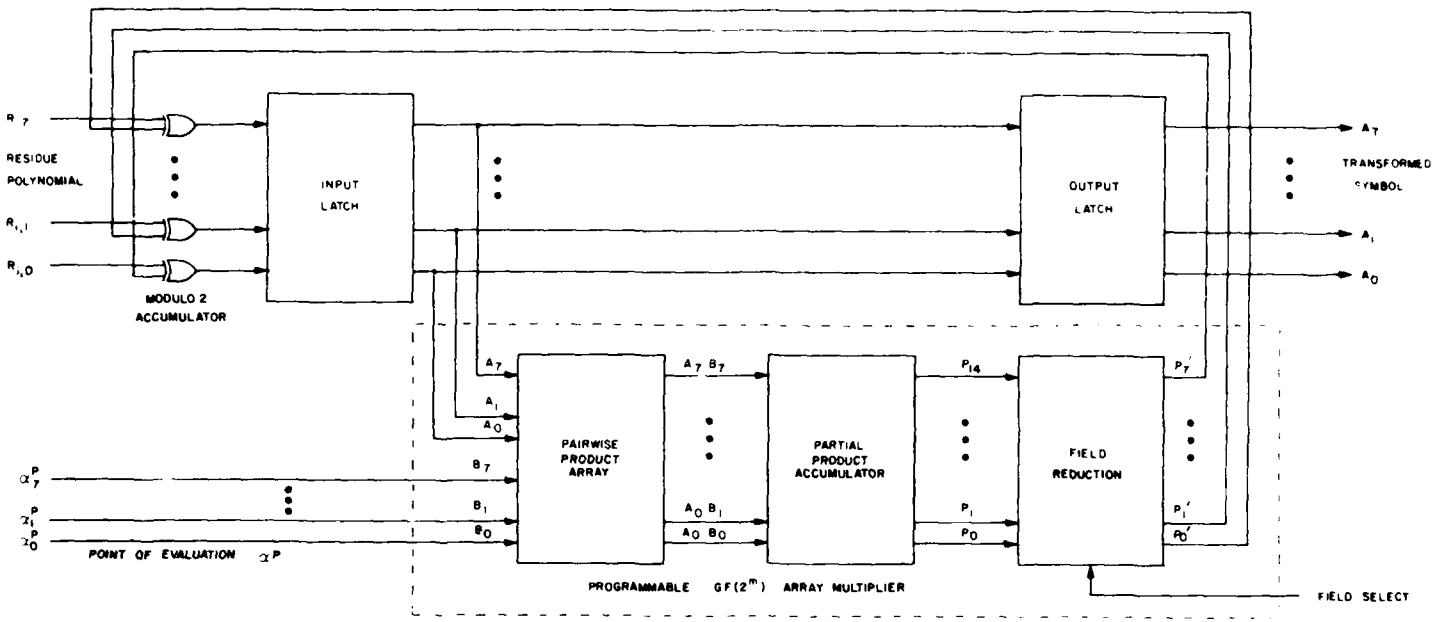


Figure 20. (51,k) Transformer's Polynomial Residue Evaluator

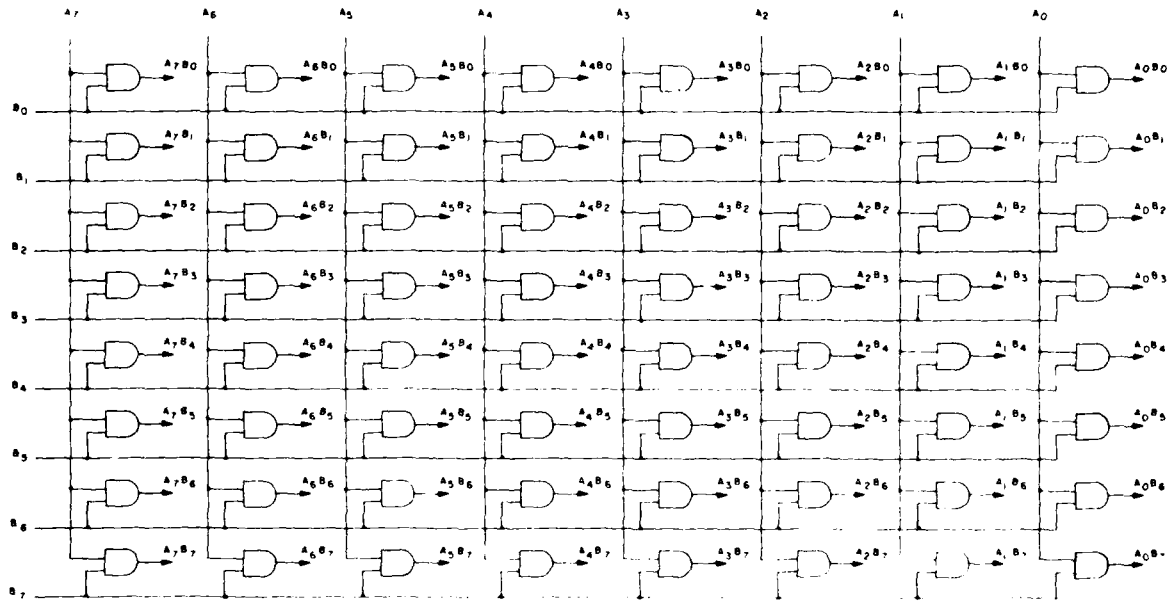


Figure 21. GF(2^m) Programmable Multiplier's Pairwise-Product Array

128 symbol error-correction slices. The breadboard can correct all combinations of t errors and s erasures provided the inequality

$$2t + s \leq n-k \leq 16 \qquad (4-1)$$

is satisfied.

The breadboard's symbol error-correction slices are implemented on four identical wire-wrap logic boards. Each of these boards contains four identical error-correction slices, each slice is equivalent to the 8-bit slice shown in Figure 17. The logic-level diagram for one of the breadboard's symbol-error correction slices is shown in Figure 22.

The field-dependent portion of the errata-location section is confined to the other two wire-wrap boards shown in the lower left corner of Figure 18. One board implements field element-division and contains a programmable $GF(2^m)$ array multiplier that is identical to the multiplier implemented in the polynomial residue evaluator. The errata-locator's sixth wire-wrap board contains the programmable $GF(2^m)$ serial multipliers that are required to implement the decoding algorithm.

The timing and control circuitry required to implement the errata-location algorithm is located behind the front panel shown in Figure 18. Also located behind this front panel are interface and self-testing circuits. The timing, control, interface and self-testing circuits are implemented in discrete combinational logic and memory.

4.3 Operational Characteristics

The operation of the breadboard is similar to the operation of the (255,k) encoder and decoder (section 3.3). The breadboard is

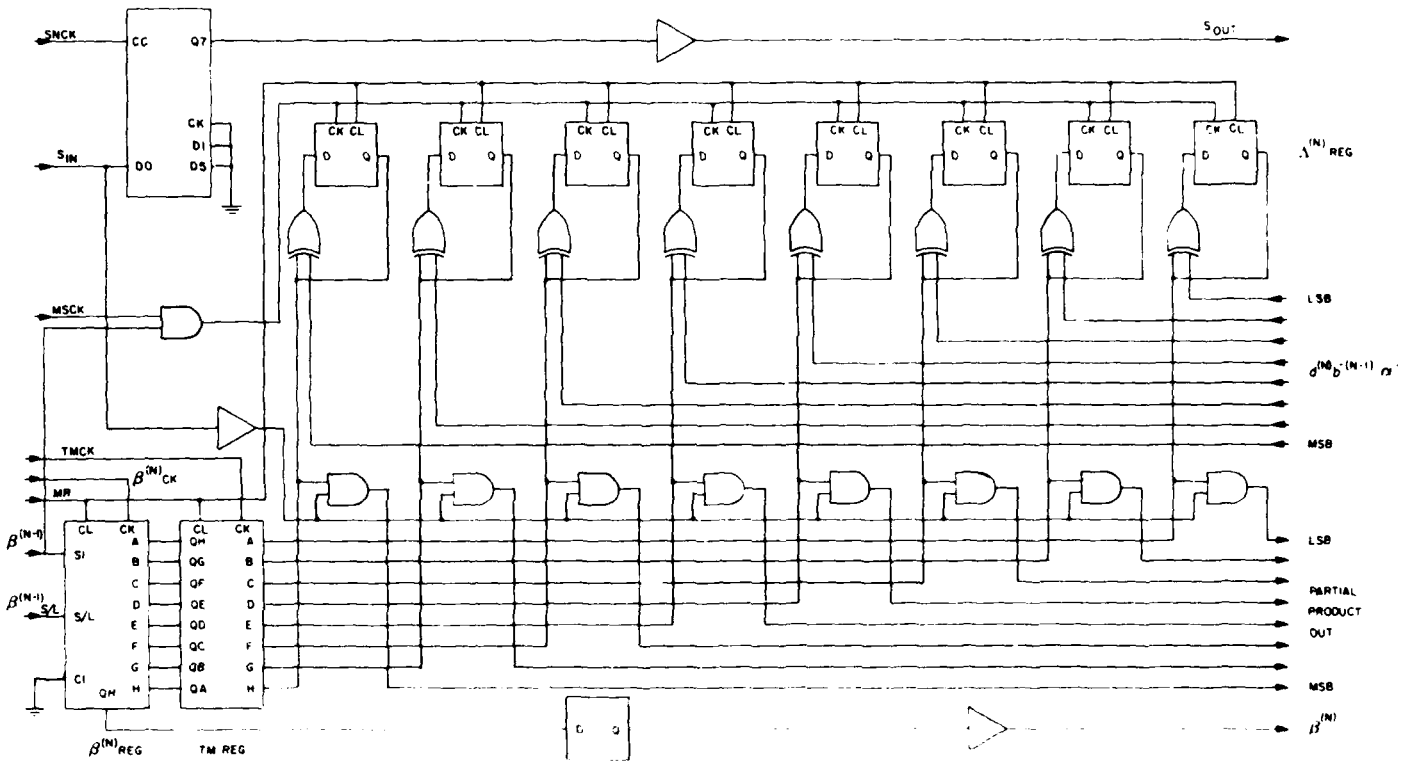


Figure 22. Schematic: 8-Bit Symbol Correction Slice

designed using readily available Schottky TTL logic. The majority of the logic functions are implemented using small-scale integrated (SSI) circuit technology, with a small section of control circuitry implemented in medium scale integrated (MSI) logic and memory.

The errata-location section implements the decoding algorithm using the same definition of machine cycle as was presented in Section 3.3. There are 20 clock cycles required to implement a single machine cycle. The time required to implement $GF(2^m)$ multiplication is the critical factor that is used to define a clock cycle. A programmable $GF(2^m)$ array multiplier, designed in Schottky TTL logic, can multiply two field elements in 60 nanoseconds. A clock cycle for the breadboard is defined to be 100 nanoseconds and a breadboard machine cycle is defined to be 2 microseconds.

The transform section can compute an n-point transform in 4n microseconds. The first 2n microseconds are required to implement the polynomial division associated with the fast polynomial evaluation algorithm. The second 2n microseconds are required to evaluate the residue polynomials. A single point in the transform is calculated during every machine cycle associated with the second 2n microseconds.

The errata-location section requires sequential data from the transformer. The first transformed point is available for processing after n + 1 machine cycles. The n machine cycles used to synthesize the errata-location polynomial and recover the k information symbols are offset one machine cycle from the n machine cycles that the transformer requires for evaluation. The total time required to decode an (n,k) code is 2n + 1 machine cycles, or (4n + 2) microseconds.

The breadboard can operate in either a single-cycle or continuous mode. In the single-cycle mode the breadboard operates on a single block of data and requires 4n microseconds to encode and 4n + 2 microseconds to decode. In a continuous mode, the breadboard accepts a

new block of data at 4n microsecond intervals. After an initial delay, the offset pulse becomes transparent and a block of decoded data is available every 4n microseconds.

The (51,k) Reed-Solomon transform encoder and decoder TTL breadboard has been interfaced with a semi-automated testing facility. The basis of this testing facility is a dedicated Hewlett-Packard 2115 minicomputer. As peripherals, the minicomputer has a CRT terminal, a floppy disk, a high-speed word generator, and a high-speed input/-output interface system. This semi-automated testing facility is shown in Figure 23. This facility was used to debug and exercise the (51,k) encoder and decoder breadboard.

4.4 Hardware Complexity

The breadboard's transform section occupies the five wire-wrap cards shown in the lower right section of the breadboard's card cage (Figure 18). This section consists of five 8" x 8" wire-wrap logic boards. Two different board designs implement the transform section. Four transformer boards implement the polynomial divider circuits, as indicated in Table XI. These boards are identical, each board containing two slices of each divider circuit. The fifth board in the transform section implements the continued product expansion for polynomial evaluation. This board contains a programmable $GF(2^m)$ array multiplier and accumulator structure.

Each of the transform's polynomial division boards contains 128 bits of shift register, and approximately 750 logic gates. The polynomial residue evaluator board contains approximately 900 logic gates. The breadboard's transformer has a total of 512 bits of shift register and 4k logic gates. Each wire-wrap board carries approximately 50 ICs so that approximately 250 ICs are used in the construction of the transformer. The logic for controlling the transformer is contained in the timing, control and interface section.

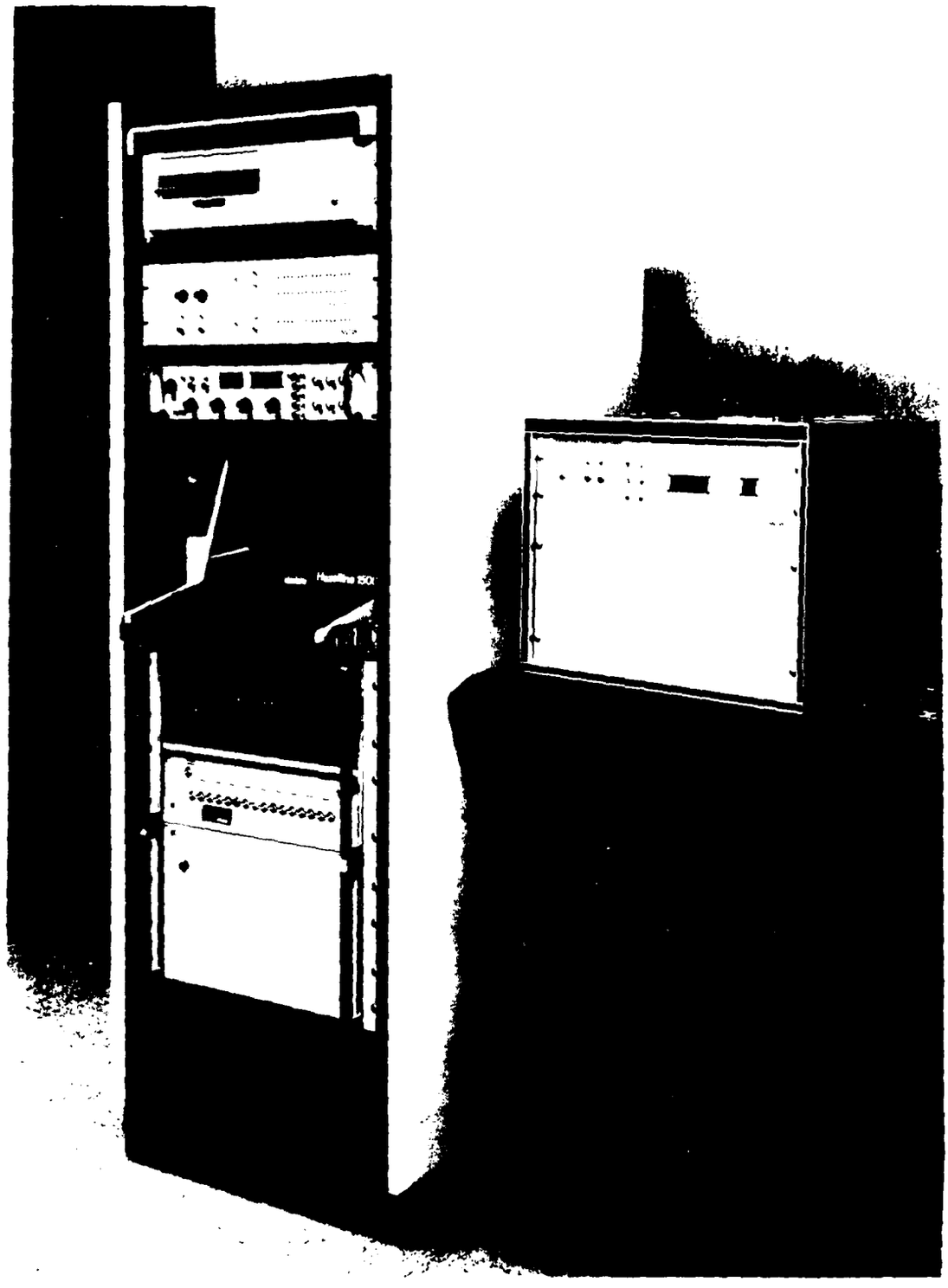


Figure 23. Semi-Automated Testbed

The errata-locator occupies the six wire-wrap cards shown in the lower left section of the card cage (Figure 18). This section consists of six 8" x 8" wire-wrap logic boards and approximately 300 ICs. All control for the errata-locator is provided by the timing, control, and interface section.

Three different logic board designs implement the errata-location section. The field-independent portion of the errata locator's architecture consists of slices of hardware that are shown in Figure 22. Four of these slices are designed to fit on one wire wrap logic board. Sixteen slices are required to implement the errata-locator. Four of the six logic boards are designed and built identically. The fifth board contains all the field-dependent logic associated with calculating the present discrepancy. This board also contains the field-dependent logic required to sequentially revise the present feedback connection polynomial. The sixth board contains the field-element division circuitry. The critical structure located on the sixth board is a programmable $GF(2^m)$ array multiplier designed identically with the multiplier used in the transformer's polynomial residue evaluator.

Each of the symbol-correction-slice boards contain 128 bits of shift register and 400 logic gates. The field dependent serial multiplier board contains approximately 1k logic gates, and the field element division board contains approximately 1.5k logic gates. The errata-location section contains a total of 512 bits of shift register and approximately 4k logic gates.

The timing, control, and interface section is located behind the front panel. This special-purpose circuitry is not repetitive. The construction of this section requires approximately 140 SSI and MSI circuits. This section provides all of the timing and control signals needed to operate the transform section. Included in these signals is the information that determines which residue is selected,

the field element at which the selected residue polynomial is to be evaluated, the order of evaluation, and all necessary clocking signals required to operate both the divisor circuits and the residue-evaluator circuit.

The timing, control and interface also supplies the breadboard's errata-location section with its timing and control signals. In addition to providing the signals required to calculate each step in the decoding algorithm, the timing and control section analyzes each step in the modified Berlekamp-Massey algorithm and dictates the necessary branching. The timing and control section performs the bookkeeping and decision making associated with the algorithm.

APPENDIX A

MULTIPLICATION IN $GF(2^m)$: ALGORITHMS AND STRUCTURES

Decoding algorithms for algebraic error-correction codes require arithmetic operations that are defined on the finite algebraic fields in which the codes are defined. The essential operations are finite-field multiplication, addition, and inversion. The effective hardware implementation of an error-correction decoder requires the design of circuitry that implements these operations.

Important algebraic error-correction codes are those that are defined over the binary extension fields $GF(2^m)$. These codes have symbols represented as binary vectors. Their encoding and decoding algorithms can be interpreted as special purpose digital signal processing algorithms. When the fields of operation are binary extension fields, finite-field addition is defined as bit-by-bit modulo-two addition and it can be implemented using Exclusive-OR circuitry. Binary extension field multiplication has many structural interpretations, each leading to a different hardware implementation, and the "best" implementation depends upon the particular application.

This appendix describes algorithms and structures that can be used to implement binary extension field multiplication. First, binary extension field multiplication is described. Then, an overview of different $GF(2^m)$ multiplier structures is presented. Finally, the multipliers that are used in the Reed-Solomon error-correction encoder and decoder are described in detail.

A.1 MULTIPLICATION IN $GF(2^m)$

A binary extension field, or Galois field, $GF(2^m)$ is a finite algebraic field that contains $2^m - 1$ nonzero field elements. The

field is generated by an m-th degree irreducible polynomial p(x), having a root α which lies in the extension field. The specific polynomial p(x) chosen for each of the fields over which the decoder operates is a primitive polynomial, meaning that the root α is a primitive field element, which in turn means that each of the nonzero field elements can be represented as a power of it (i.e., α^0, α^1, α^2, ..., α^{2^m-2}). Each nonzero field element can also be represented as a binary m-tuple which can be considered a vector relative to the normal basis $\{\alpha^0, \alpha^1, \ldots, \alpha^{m-1}\}$. This multiplication of two nonzero field elements can be implemented using either representation. The addition of two field elements is conveniently implemented as vector addition.

Binary extension field addition can be interpreted as the pairwise modulo-2 addition of the m-tuple representation of the field elements to be added

$$
\begin{aligned}
&\alpha^i,\ \alpha^j \in GF(2^m) \qquad\qquad 0 \le i,\ j \le 2^m-2 \\
&\alpha^i = \alpha^i_{m-1}, \ldots, \alpha^i_1, \alpha^i_0 \\
&\alpha^j = \alpha^j_{m-1}, \ldots, \alpha^j_1, \alpha^j_0 \\
&\alpha^i + \alpha^j \triangleq \alpha^i_{m-1} \oplus \alpha^j_{m-1}, \ldots, \alpha^i_1 \oplus \alpha^j_1, \alpha^i_0 \oplus \alpha^j_0
\end{aligned}
\qquad \text{(A-1)}
$$

Multiplication of two field elements that are represented as powers of the primitive element has a familiar logarithmic appearance. Binary extension field multiplication using this symbolic representation has a compact form and it is well suited for implementation using table look-up procedures.

$$\alpha^i, \alpha^j \in GF(2^m) \qquad 0 \le i, j \le 2^m-2$$

$$\alpha^i \cdot \alpha^j \triangleq \alpha^{(i+j) \bmod (2^m-1)} \qquad \text{(A-2)}$$

Binary extension field multiplication using field elements re-represented as m-tuples has a definition that resembles convolution or polynomial multiplication.

$$\alpha^i, \alpha^j \in GF(2^m) \qquad 0 \le i, j \le 2^m-2$$

$$\alpha^i = (\alpha^i_{m-1}, \ldots, \alpha^i_1, \alpha^i_0) \qquad \alpha^i_\ell = 0 \quad \ell < 0, \ \ell > m-1$$

$$\alpha^j = (\alpha^j_{m-1}, \ldots, \alpha^j_1, \alpha^j_0) \qquad \alpha^j_\ell = 0 \quad \ell < 0, \ \ell > m-1 \qquad \text{(A-3)}$$

$$\alpha^i \cdot \alpha^j \triangleq \left\{ \sum_{\ell=0}^{2(m-1)} \left\{ \sum_{n=0}^{m-1} \alpha^j_{\ell-n} \, \alpha^i_n \right\} x^\ell \right\} \bmod (p(x))$$

This form of multiplication can be interpreted as a two-step procedure. First, two polynomials of degree at most m-1 are multiplied to form a product polynomial of degree at most 2m-2. Secondly, the product polynomial is reduced, modulo p(x), to a polynomial of degree less than or equal to m-1 and whose coefficients are the product m-tuple. The latter definition of binary extension field multiplication, (equation A-3), can be expanded to indicate the intermediate operations that are required for implementation.

$$\alpha^i,\ \alpha^j \in GF(2^m) \qquad 0 \le i,\ j \le 2^m - 2$$

$$\alpha^i \cdot \alpha^j \triangleq \left\{ \sum_{\ell=0}^{2(m-1)} \left\{ \sum_{n=0}^{m-1} \alpha^j_{\ell-n}\ \alpha^i_n \right\} x^\ell \right\} \bmod p(x)$$

$$= \Big\{ (\alpha^j_0\ \alpha^i_0)x^0 + (\alpha^j_1\ \alpha^i_0 \oplus \alpha^j_0\ \alpha^i_1)x^1 +$$

$$\vdots \qquad \text{A-4}$$

$$+ (\alpha^j{}_{m-1}\ \alpha^i_0 \oplus \alpha^j{}_{m-2}\ \alpha^i_1 \oplus \cdots \oplus \alpha^j_0\ \alpha^i{}_{m-1})x^{m-1} +$$

$$\vdots$$

$$+ (\alpha^j{}_{m-1}\ \alpha^i{}_{m-1})x^{2(m-1)} \Big\} \bmod p(x)$$

There are three steps used to implement equation (A-4). First, the pairwise product of each term within the two m-tuples to be multiplied is formed. Next, these pairwise products are accumulated to form the partial products that represent the coefficient of the product polynomial. Finally, the product polynomial is reduced modulo p(x).

A.2 $GF(2^m)$ MULTIPLIER STRUCTURES

Binary extension field multiplier structures can be partitioned into two classes. One class is based upon table look-up procedures and its hardware implementations are memory intensive. The other class of multiplier structures is based upon the algebraic properties

of the binary extension fields and the hardware implementations use random logic. Both implementation strategies have their particular advantages and disadvantages.

Memory Intensive Multiplier Structures

The simplest form of a $GF(2^m)$ multiplier implements a direct table look-up procedure. There are many possible variations on this strategy, but in general the two field elements to be multiplied are used to identify a particular memory location in which the precalculated product is stored. The different implementations using this strategy depend on the ways in which the elements to be multiplied can be combined to identify the memory location and the ways in which the computed product element can be stored.

Memory intensive multiplier structures have common characteristics. Since these multipliers are basically memory, the complexity of the resulting hardware implementation is dependent on the selected device technology. Because of the range of available memory technologies, these multipliers can have a wide range of operational rates. A disadvantage of memory-intensive multipliers is that the storage requirements increase exponentially with the degree of the field extension. Available memory technologies can provide very fast (<100 nanosecond) multiplication times for small fields ($m \leq 5$), but access times increase rapidly as the fields become larger.

Random-Logic Multiplier Structures

Random-logic multiplier structures separate into two different implementation classes. The computational steps outlined in equation (A-4) can be performed serially in time, and the resulting hardware implementation uses sequential logic. Alternatively, the necessary calculations can be performed concurrently in time, and the resulting implementation uses arrays of combinational logic. These two

approaches have a unique relationship. Sequential multipliers perform their computational steps in series, often using the same hardware to compute different steps. Therefore, sequential multipliers tend to have simple structures, (i.e., LFSRs), but their multiplication times are relatively slow. The combinational logic array multipliers perform many operations simultaneously, using different sections of hardware to compute different steps. The array-type multipliers tend to have complex hardware but their multiplication times are very fast. In general, there is an inverse relationship between a multiplier structure's hardware complexity and its multiplication time.

A $GF(2^m)$ sequential multiplier is shown in Figure A-1. This circuit directly implements the computational steps of equation (A-3). The sequential multiplier is a simple structure, requiring 4m bits of shift-register circuitry and a maximum of 12m logic gates (a logic gate is taken to mean either a two input NAND or NOR gate). A complete multiplication cycle, assuming serial output, takes 2m-1 clock cycles.

A simple description will illustrate this circuit's operation. Initially, the irreducible polynomial associated with the field of operation, $p(x) = p_0 + p_1x + \ldots + p_mx^m$, is loaded into the P register. The product register, C, is cleared to zero. The m-tuple representation of the field element α^i is stored in the multiplier register, A, while the m-tuple representation of the field element α^j is stored in the multiplicand register, B. The multiplier register effectively holds α^i as the coefficients of an (m-1)th degree polynomial. The first clock pulse latches $\alpha^j{}_{m-1}(\alpha_0{}^i + \alpha_1{}^i x + \cdots + \alpha^i{}_{m-1}x^{m-1})$ into the product register. On the next clock cycle this product is shifted toward the right. This is equivalent to multiplying the

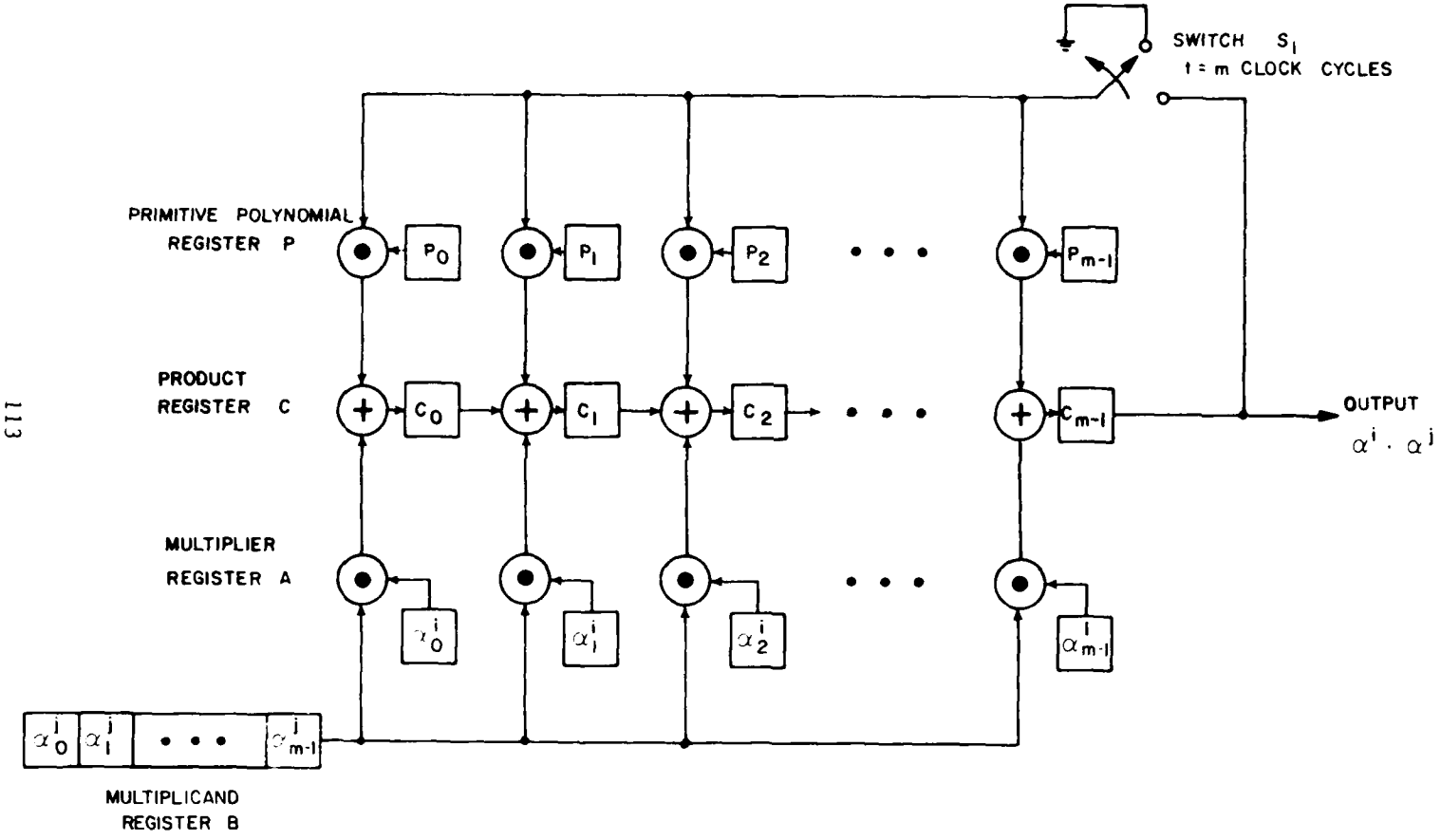


Figure A-1. Sequential GF(2^m) Multiplier

product with x. Prior to this shift, the contents of the C_{m-1} register was $\alpha^{j}_{m-1}\,\alpha^{j}_{m-1}$. This corresponds to $\alpha^{j}_{m-1}\,\alpha^{i}_{m-1}x^{m-1}$. The shift creates $\alpha^{j}_{m-1}\,\alpha^{j}_{m-1}x^{m}$, and the feedback connection polynomial, p(x), performs division or polynomial modulo reduction on this overflow term. At the conclusion of the second clock cycle, the product register contains

$$\alpha^{j}_{m-1}(\alpha^{i}_{0} + \alpha^{i}_{1}x + \cdots + \alpha^{i}_{m-1}x^{m-1})x \mod p(x) + \alpha^{j}_{m-2}(\alpha^{i}_{0} + \alpha^{i}_{1}x + \cdots + \alpha^{i}_{m-1}x^{m-1})x \mod p(x) \tag{A-5}$$

This process continues for m-1 clock cycles. At the completion of the (m-1)th clocking cycle, the contents of the product register is:

$$\begin{aligned}(\cdots\ ((&\alpha^{j}_{m-1}(\alpha^{i}_{0} + \alpha^{i}_{1}x + \cdots + \alpha^{i}_{m-1}x^{m-1})x \mod p(x)\\ &+ \alpha^{j}_{m-2}(\alpha^{i}_{0} + \alpha^{i}_{1}x + \cdots + \alpha^{i}_{m-1}x^{m-1}))x \mod p(x)\\ &\quad\vdots\\ &+ \alpha^{j}_{1}(\alpha^{i}_{0} + \alpha^{i}_{1}x + \cdots + \alpha^{i}_{m-1}x^{m-1}))\cdots)x \mod p(x)\\ &+ \alpha^{j}_{0}(\alpha^{i}_{0} + \alpha^{i}_{1}x + \cdots + \alpha^{i}_{m-1}x^{m-1})\end{aligned} \tag{A-6}$$

This expression can be reduced to:

$$(\alpha^{j}_{m-1}x^{m-1} + \alpha^{j}_{m-2}x^{m-2} + \cdots + \alpha^{j}_{0})(\alpha^{i}_{0} + \alpha^{i}_{1}x + \cdots + \alpha^{i}_{m-1}x^{m-1}) \bmod p(x)$$

$$= \left\{ \sum_{\ell=0}^{2(m-1)} \alpha^{i}_{\ell} \left(\sum_{n=0}^{m-1} \alpha^{j}_{\ell-n} \right) x^{\ell} \right\} \bmod p(x)$$

At the conclusion of the (m-1)th clocking cycle, the switch S_1 is grounded and the product is shifted out of the product register on the following m clock cycles. There are 2m-1 clock cycles required for complete multiplication.

The structure of a GF(2^m) array multipler is shown in Figure A-2. It implements the computational steps outlined in equation (A-4). The array multiplier consists of three functionally separate hardware sections. The first section calculates the m^2 pairwise products between the m-tuples associated with the field elements to be multiplied. A second section operates on these products and accumulates the 2m-1 terms that are the coefficients of the product polynomial indicated in equation (A-4). The third section operates on the 2m-1 accumulated product terms and implements modulo p(x) reduction, resulting in the final product.

The array multiplier can multiply two field elements quickly. Fast multiplication rates are obtained because the added circuitry is included to perform the requisite operations in parallel and because the modulo p(x) computation is implemented asynchronously as an end-around-carry operation. The end-around-carry reduction is implemented by simply feeding back the overflow bits from the product equation in a predetermined manner.

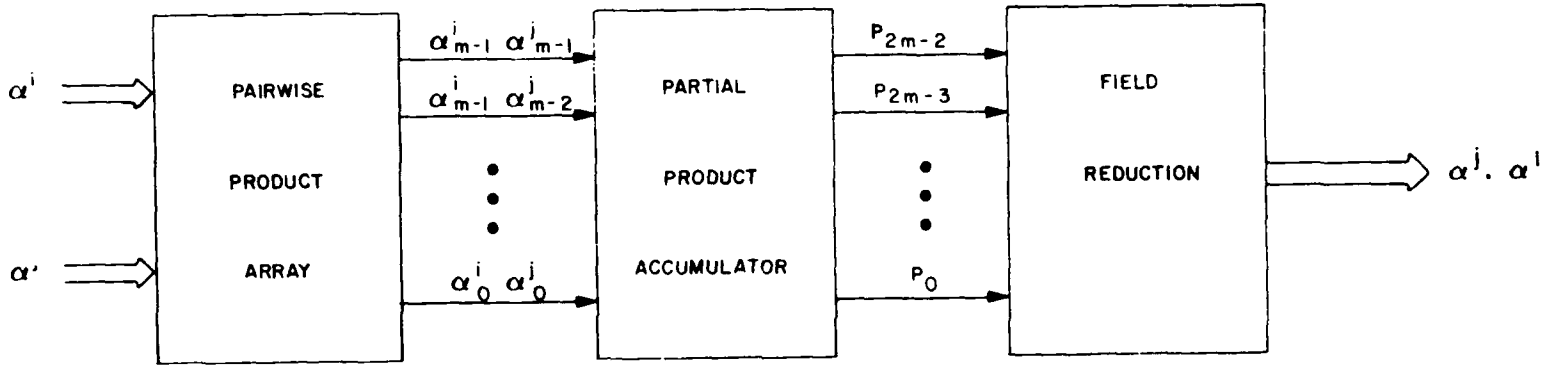


Figure A-2. GF(2^m) Array Multiplier

The hardware complexity of the array multiplier shown in Figure A-2 can be measured in terms of logic gates. The pairwise product array requires m^2 gates to calculate the m^2 partial products. The accumulator array is configured as 2m-1 Exclusive-OR trees. The maximum number of logic gates required to implement this section in m(m-1) gates. The field reduction circuitry can be implemented as m Exclusive-OR trees, with the maximum number of logic gates equal to m^2. The total number of gates required to implement the $GF(2^m)$ array multiplier is on the order of $4m^2$ gates.

The total time required to multiply two field elements is dependent upon the propagation delay, τ, through a logic gate. The time required to calculate the m^2 pairwise products is 2τ. The delay through either the accumulator array or the field reduction circuitry is $3\tau\lceil\log_2(m)\rceil$, where $\lceil x \rceil$ is the smallest integer larger than x. The maximum time required to multiply two field elements in $GF(2^m)$ is $2\tau + 6\tau\ [\text{Log}_2(m)]$. Assuming a 3 nanosecond propagation delay, two field elements from $GF(2^8)$ can be multiplied in 60 nanoseconds.

A.3 REED-SOLOMON ENCODER AND DECODER MULTIPLIER STRUCTURES

The design for the Reed-Solomon (255,k) transform encoder and decoder has five separate requirements for binary extension field multiplication. Each application requires multiplication in $GF(2^m)$ where m = 4,5,6,7, or 8. The design of a multiplier that can be programmed to operate in more than one extension field is difficult because the multiplier structure that is designed to operate in $GF(2^m)$ is not directly expandable for operation in $GF(2^{m+1})$. The requirement to operate in five different fields eliminates the memory intensive multipliers as candidate multiplier structures since need for reconfigurability produces multiplication rates that are too slow.

Three field-dependent multiplier structures are used in the encoder and decoder; two of the designs are repeated twice. The design of each multiplier is based on the multiplication algorithm shown in equation (A-3). The primitive polynomials that generate each field are shown in Table A-I. The remainder of this appendix will describe the design and operation of the three field-dependent multipliers used in the encoder and decoder.

$GF(2^m)$ Programmable Array Multiplier

The most complicated multiplier structure used in the encoder and decoder is a programmable $GF(2^m)$ array multiplier similar to the one previously described. (A block diagram of the programmable array multiplier is shown in Figure A-3). The programmable array multiplier consists of a pairwise product array, an accumulation array and field reduction circuitry. The field reduction circuitry is reconfigurable to provide modular polynomial reduction using any of the primitive polynomials shown in Table A-I.

The programmable array multiplier is designed to operate with our standard 8-bit symbols so that any symbol from $GF(2^m)$, with $m < 8$, has zeros padded in the most significant bit positions. The programmable array multiplier's pairwise product array operates with two 8-bit symbols and calculates 64 pairwise products. The padded zeros in the standard 8-bit symbols produce the correct zero products for operation in fields with $m < 8$. The pairwise product array contains no additional hardware than would be required for normal operation in $GF(2^8)$.

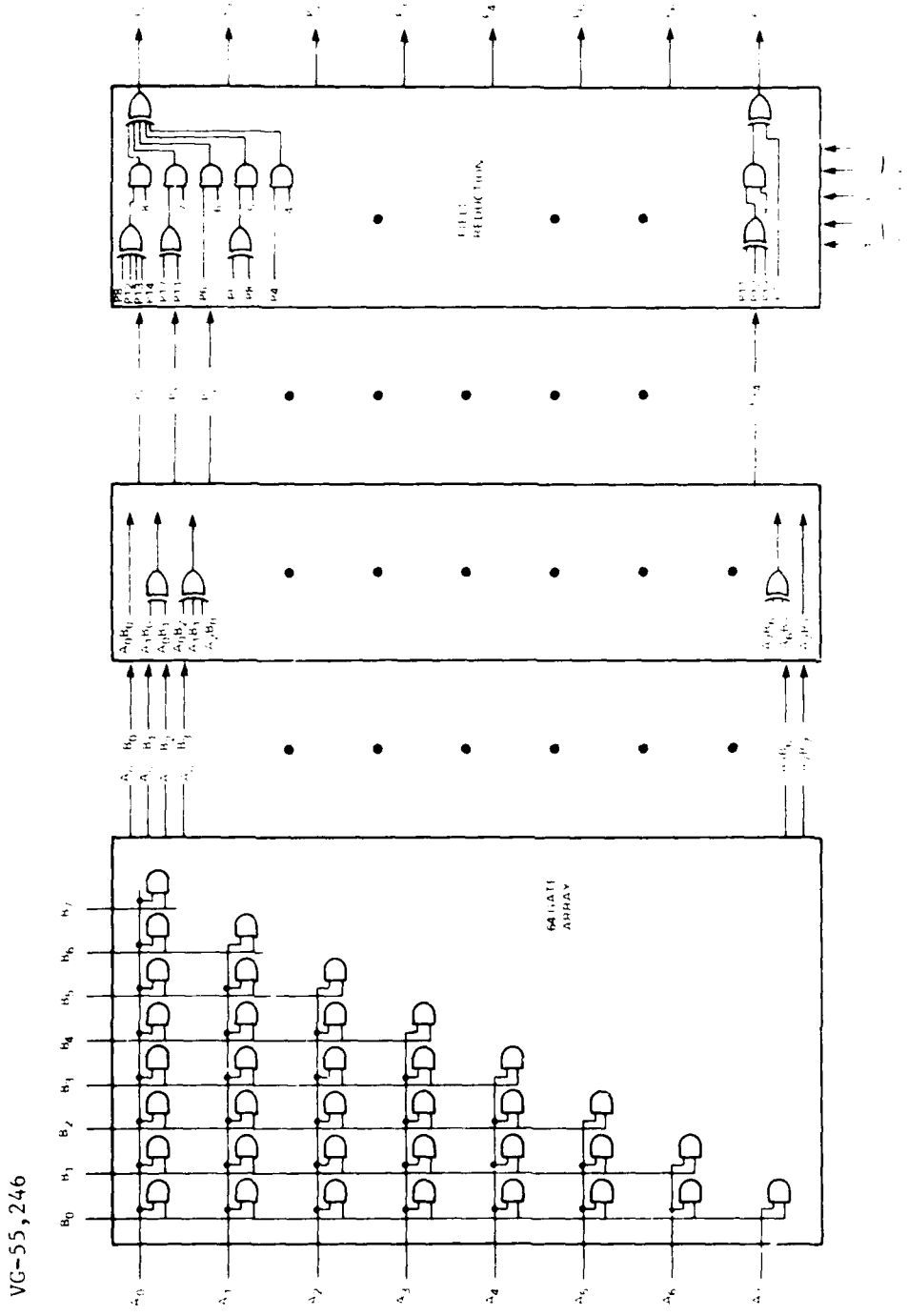
The accumulator array operates on the 64 pairwise products from the product array to calculate 15 partial sums. These terms are the coefficients of the product polynomial shown in equation (A-4). The 15 coefficients are formed using 15 Exclusive-OR trees. When multiplication is required in fields where $m < 8$, the padded zeros that

Table A-I

Primitive Polynomials Used to Design the

Programmable $GF(2^m)$ Multiplier Structures

Extension Field	Primitive Polynomial
$GF(2^4)$	$P_4(x) = x^4 + x + 1$
$GF(2^5)$	$P_5(x) = x^5 + x^2 + 1$
$GF(2^6)$	$P_6(x) = x^6 + x + 1$
$GF(2^7)$	$P_7(x) = x^7 + x^3 + 1$
$GF(2^8)$	$P_8(x) = x^8 + x^4 + x^3 + x^2 + 1$



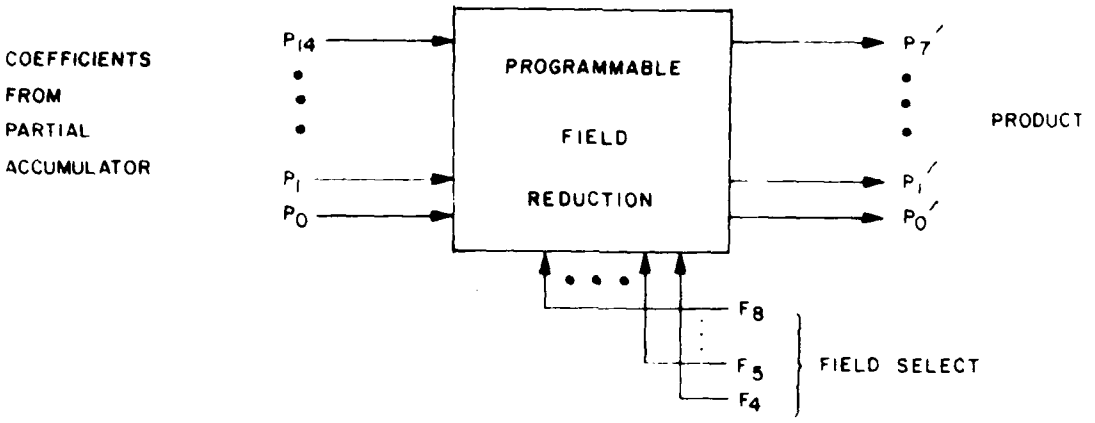
VG-55,246

Figure A-3. Programmable CFA[m] Array Multiplier

produce the correct zero-valued pairwise products result in the correct summations within the accumulator array. Operation in the five different finite fields is obtained without an increase in the hardware that is required for operation in $GF(2^8)$.

The field reduction circuitry is the only field-dependent section of the programmable array multiplier. This structure uses the 15 partial sums that have been calculated in the accumulator and implements polynomial reduction modulo p(x). The result is a standard 8-bit symbol that is the correct product. The field-reduction circuit implements asynchronous end-around-carry to compute modular polynomial reduction. The reduction is different for each field of operation because a different primitive polynomial p(x) is used. Programmability is provided by designing different feedback paths to be selected for the different fields. A block diagram of the programmable field reduction circuit is shown in Figure A-4. Here $\{P'_0, P'_1, \ldots, P'_7\}$ are the eight bits in our standard symbol representation of the product, and $\{P_0, P_1, \ldots, P_{14}\}$ are the coefficients of the product polynomial formed in the accumulator array. The signals $\{F_4, F_5, F_6, F_7, F_8\}$ represent control flags that indicate the field of operation. For multiplication in $GF(2^m)$, the signal F_m is a logic "1"; all other F_i's, where $i \neq m$, are set to logic "0". These flags reconfigure the polynomial-reduction circuitry to the correct feedback paths. The logic functions implemented by the field reduction circuit are shown in Figure A-4.

The field-reduction array consists of 8 Exclusive-OR trees. Each tree has inputs that are gated with the field-select control signals F_m. For operation in $GF(2^m)$, each tree accumulates only the terms in the logic equations shown in Figure A-4 that are associated with the field-select signal F_m.



$$P_0' = P_0 + P_4F_4 + (P_5+P_8)F_5 + (P_6)F_6 + (P_7+P_{11})F_7 + (P_8+P_{12}+P_{13}+P_{14})F_8$$

$$P_1' = P_1 + (P_4+P_5)F_4 + P_6F_5 + (P_6+P_7)F_6 + (P_8+P_{12})F_7 + (P_9+P_{13}+P_{14})F_8$$

$$P_2' = P_2 + (P_5+P_6)F_4 + (P_5+P_7+P_8)F_5 + (P_7+P_8)F_6 + P_9F_7 + (P_8+P_{10}+P_{12}+P_{13})F_8$$

$$P_3' = P_3 + P_6F_4 + (P_6+P_8)F_5 + (P_8+P_9)F_6 + (P_7+P_{10}+P_{11})F_7 + (P_8+P_9+P_{11}+P_{12})F_8$$

$$P_4' = P_4\bar{F}_4 + P_7F_5 + (P_9+P_{10})F_6 + (P_8+P_{11}+P_{12})F_7 + (P_8+P_9+P_{10}+P_{14})F_8$$

$$P_5' = P_5(\bar{F}_4 \cdot \bar{F}_5) + P_{10}F_6 + (P_9+P_{12})F_7 + (P_9+P_{10}+P_{11})F_8$$

$$P_6' = P_6(F_7+F_8) + P_{10}F_7 + (P_{10}+P_{11}+P_{12})F_8$$

$$P_7' = (P_7+P_{11}+P_{12}+P_{13})F_8$$

Figure A-4. Programmable GF(2^m) Array Multiplier Field Reduction Circuit

The $GF(2^m)$ programmable array multiplier is the critical component within the transformer's polynomial residue evaluator. The array multiplier was selected for this application because of its fast multiplication rates. The multiplier is also attractive because of its repetitive architecture and the low complexity of hardware required for reconfigurability.

The programmable array multiplier is also used in the implementation of the Berlekamp-Massey shift register synthesis algorithm. The array multiplier was chosen for this application because of its short processing time. The array multiplier calculates the product of the present discrepancy and the inverse of the past discrepancy $(d^{(N)}{}_b{}^{-(N-1)})$. This product is calculated once during each iteration of the algorithm and the time required to form this product limits the operational speed of the entire errata-location section.

$GF(2^m)$ α^2 Structure

The second field-dependent multiplier structure used in the implementation of the encoder and decoder is a special-purpose, field-element squaring circuitry. This structure calculates the squared product of any field element in $GF(2^m)$, where m = 4,5,6,7, or 8. The α^2 multiplier uses our standard symbol representation and implements

$$\alpha^i \cdot \alpha^i = \alpha^{2i \bmod (2^m-1)}$$

$$\alpha^{2i} = \sum_{j=0}^{7} \alpha_j^i \left(\sum_{k=0}^{7} \alpha_k^i x^k \right) x^j \bmod p(x) \qquad \text{(A-8)}$$

where α^i is an element from $GF(2^m)$ and p(x) is the associated primitive polynomial shown in Table A-I. In equation (A-8), the cross-

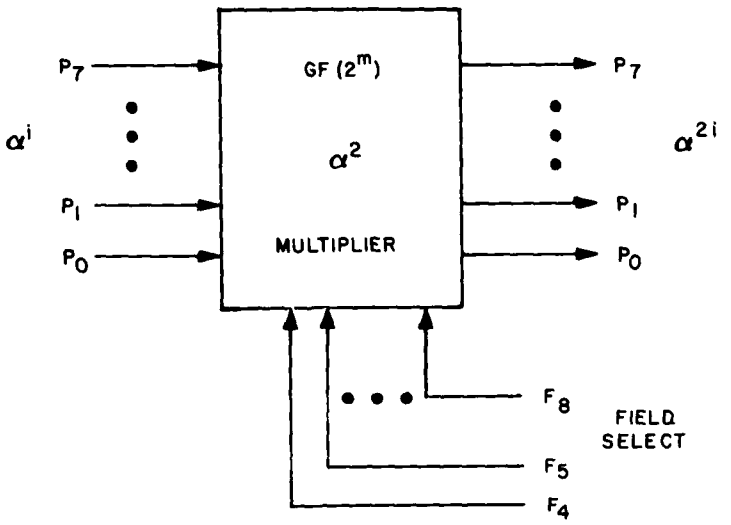
products $\alpha_j^i \alpha_k^i$, such that $k \neq j$, are zero modulo-2. The products $\alpha_j^i \alpha_k^i$, such that $k=j$, are α_j^i. Only the even ordered coefficients of the product polynomial are formed

$$\alpha^i \cdot \alpha^i = (\alpha_0^i + \alpha_1^i x^2 + \alpha_2^i x^4 + \alpha_3^i x^6 + \alpha_4^i x^8 + \alpha_5^i x^{10} + \alpha_6^i x^{12} + \alpha_7^i x^{14}) \mod p(x) \qquad (A-9)$$

The programmable field reduction circuitry that implements equation (A-9) is similar to the circuitry used in the programmable array multiplier. However, the field reduction associated with the α^2 multiplier is simpler because there are no odd ordered coefficients in the product polynomial. The i-th bit in the field element to be squared is the 2i-th bit in the product polynomial making the square product formation implicit. The logic equations that implement the α^2 multiplier are shown in Figure A-5. In this figure, the input variables $\{P_0, P_1, \ldots, P_7\}$ represent the eight bits in our standard symbol representation of the field element and the variables $\{P_0', P_1', \ldots, P_7'\}$, represent the squared element. Again, the variables $\{F_4, F_5, F_6, F_7, F_8\}$ are signals that represent the desired field of operation.

Field Element Inversion Circuit

The $GF(2^m)$ programmable array multiplier and the $GF(2^m)$ α^2 multiplier can be combined to implement a division-by-inversion algorithm. Once during each iteration of the Berlekamp-Massey algorithm, the product $d^{(N)}b^{-(N-1)}$ is formed, where both $d^{(N)}$ and $b^{-(N-1)}$ are elements from $GF(2^m)$. The term $b^{-(N-1)}$ is the multiplicative inverse of the



$$P_o' = P_o + P_2F_4 + P_4F_5 + P_3F_6 + (P_4+P_6+P_7)F_8$$

$$P_1' = P_2F_4 + P_3F_5 + P_3F_6 + (P_4+P_6)F_7 + P_7F_8$$

$$P_2' = P_1 + P_3F_4 + (P_3+P_4)F_5 + P_4F_6 + P_3F_7 + (P_4+P_6)F_8$$

$$P_3' = P_3F_4 + (P_3+P_4)F_5 + P_4F_6 + P_3F_7 + (P_4+P_6)F_8$$

$$P_4' = P_2\overline{F}_4 + P_5F_6 + (P_4+P_6)F_7 + (P_4+P_5+P_6)F_8$$

$$P_5' = P_5F_6 + P_6F_7 + P_5F_8$$

$$P_6' = P_3(F_7+F_8) + P_5F_7 + (P_5+P_6)F_8$$

$$P_7' = P_6F_8$$

Figure A-5. Programmable α^2 Multiplier

field element b^{N-1}. The structure that is used to implement this product is shown in Figure A-6. The field element $b^{(N-1)}$ is represented by α^i and the field element $d^{(N)}$ is represented by α^j. The desired product is $\alpha^j \alpha^{-i}$ or α^{j-i}. The structure in Figure A-6 first calculates α^{-i} from the element α^i and then multiplies α^j times α^{-i} to complete the division process. This structure calculates the inverse of any field element from $GF(2^m)$, m = 4,5,6,7, or 8. The structure shown Figure A-6 calculates the inverse of α^i in $GF(2^m)$ by implementing

$$\begin{aligned} \alpha^{-i} &= \alpha^{i(2^m-2)} \\ &= (\alpha^{i(2^{m-1}-1)})^2 \\ &= (\alpha^i \cdot \alpha^{2i} \cdot \alpha^{4i} \cdots \alpha^{(m-2)i})^2 \end{aligned} \qquad \text{(A-10)}$$

The operation of the field-element inversion circuit can be described with the aid of an example. On the first clock cycle, switches S_1 and S_4 are closed and all other switches are open. The previous discrepancy $\alpha^i (= b^{(N-1)})$ is fed through the α^2 multiplier and the result is latched into the X information register. The element α^i is simultaneously fed into the Y formation register. The $GF(2^m)$ array multiplier asynchronously calculates the product of the Y information register (α^i) and the X information register (α^{2i}). The result is latched into the P register. At the conclusion of the first clock cycle, the P information register contains α^{3i}. During the second clock cycle switches S_2 and S_6 are closed and all other switches are open. The contents of the P register are fed directly into the Y register, while the contents of the X register

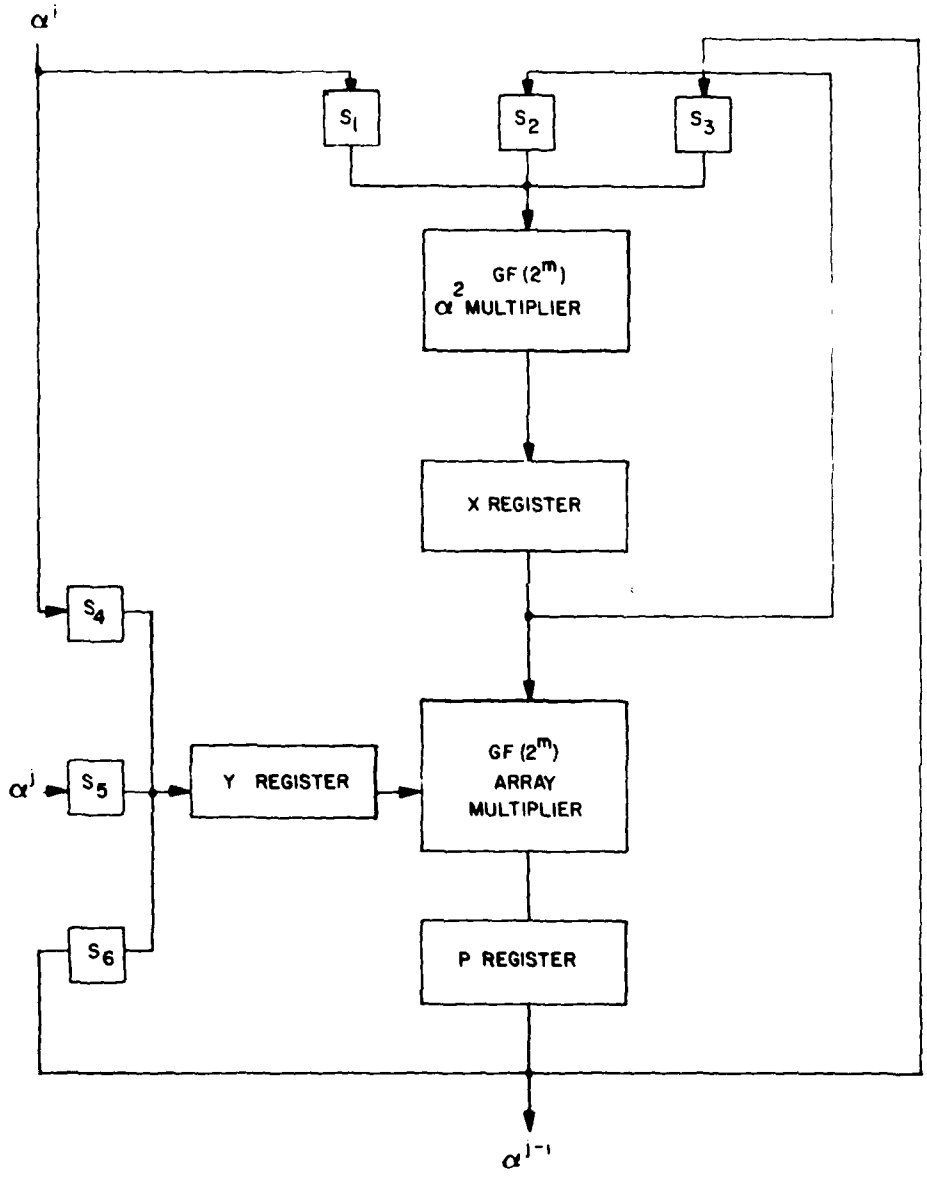


Figure A-6. Field Element Division Circuit

are fed back into the $GF(2^m)$ α^2 multiplier, and the squared product is stored in the X register. The contents of the X and Y registers are then multiplied, and the product, α^{7i}, is stored in the P register. This operation of the structure is repeated until m-2 clock cycles have been completed. At this time, the contents of the P register is $\alpha^{-i/2}$. On the next clock cycle, switches S_3 and S_5 are closed and all other switches are open. The contents of the P register are fed to the $GF(2^m)$ α^2 multipliers. The results are latched into the X register, while the present discrepancy α^j $(=d^{(N)})$, is fed into the Y register. The contents of the X and Y registers are then multiplied, and the product, α^{j-i}, is stored in the P register.

The operation of the field element inversion circuit requires m-1 clock cycles. The information shown in Table A-II indicates the status of each switch and the contents of each information register as a function of clock cycles. Table A-II represents the operation of the $GF(2^m)$ division circuit when m=8. Operation for $m < 8$ is accomplished by using fewer cycles to square the contents of the X register.

$GF(2^m)$ Serial Multiplier

The final field-dependent multiplier structure used in the encoder and decoder is one that is used for sequential multiplication. This multiplier is used in the calculation of the present discrepancy, $d^{(N)}$. The same multiplier design is used in the present feedback polynomial calculator.

The description of the $GF(2^m)$ serial multiplier is facilitated by reexamining the definition of $GF(2^m)$ multiplication as defined in equation (A-3). This definition of multiplication was shown to be equivalent to the multiplication of two (m-1)th degree polynomials followed by the polynomial reduction of the resulting 2(m-1)th degree

Table A-II

Field Element Division in $GF(2^8)$

Clock Cycle	X Register	Y Register	P Register	Switches: 0 = Open, 1 = Closed S_1	S_2	S_3	S_4	S_5	S_6
Initialize	0	0	0	-	-	-	-	-	-
1	α^{2i}	α^{i}	α^{3i}	1	0	0	1	0	0
2	α^{4i}	α^{3i}	α^{7i}	0	1	0	0	0	1
3	α^{8i}	α^{7i}	α^{15i}	0	1	0	0	0	1
4	α^{16i}	α^{15i}	α^{31i}	0	1	0	0	0	1
5	α^{32i}	α^{31i}	α^{63i}	0	1	0	0	0	1
6	α^{64i}	α^{63i}	α^{127i}	0	1	0	0	0	1
7	$\alpha^{254i} = \alpha^{-i}$	α^{j}	α^{j-i}	0	0	1	0	1	0

Note: For operation in $GF(2^m)$, on the m-2 clock pulse (P Register)^2 = α^{-i}

product polynomial. The $GF(2^m)$ programmable array multiplier implements this multiplication with all products calculated simultaneously. The serial multiplier operates sequentially to implement equation (A-3) in the form

$$\alpha^i \cdot \alpha^j = \sum_{\ell=0}^{m-1} \alpha_\ell^i (\alpha^j) x^\ell \bmod p(x) \qquad \text{(A-11)}$$

This equation is implemented using the structure shown in Figure A-7, which uses m clock cycles to complete the calculation. Prior to the first clock cycle, all latches have been cleared to zero, switch S_1 has been closed and switch S_2 opened. The first clock cycle corresponds to $\ell=0$ in equation (A-11). During this cycle α^j is fed through switches, and latched in the information register X. The contents of X are multiplied by α_0^i and latched in register P. During the second machine cycle, $\ell=1$, switch S_1 is opened and switch S_2 is closed. The contents of the X register (α^j) are fed to the serial multiplier, and the output of the multiplier ($\alpha^j x \bmod p(x)$) is latched into the X register. The contents of the X register are then multiplied with α_1^i and accumulated with the contents of the P register. After two full clock cycles the content of the P register is $\alpha^j\alpha_0^i + \alpha^j\alpha_1^i x \bmod p(x)$. The operation continues for m complete clock cycles, after which the P register contains the product $\alpha^{i,j}$.

The critical component of the multiplier structure shown in Figure A-7 is the $GF(2^m)$ serial multiplier. This multiplier takes any field element α^j and forms the field element $\alpha^j x \bmod p(x)$ which is equivalent to α^{j+1}. A diagram of the programmable $GF(2^m)$ serial multiplier is shown in Figure A-8. In this figure, the variables $\{P_0, P_1, \ldots, P_7\}$ represent our standard symbol representation of the

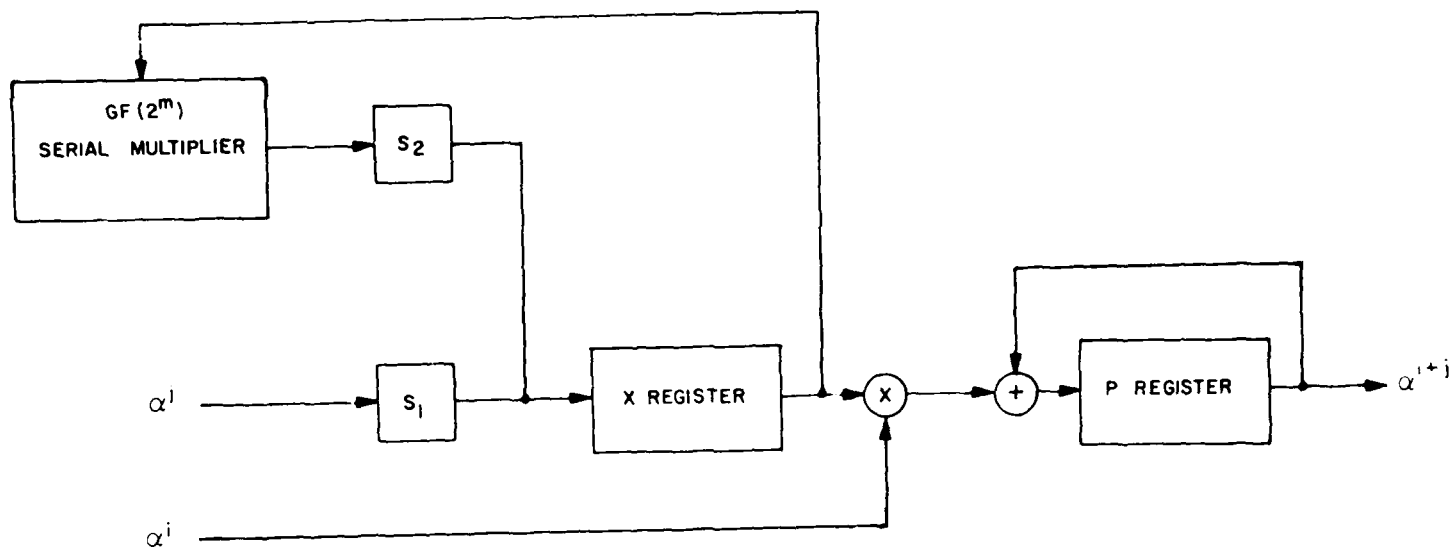
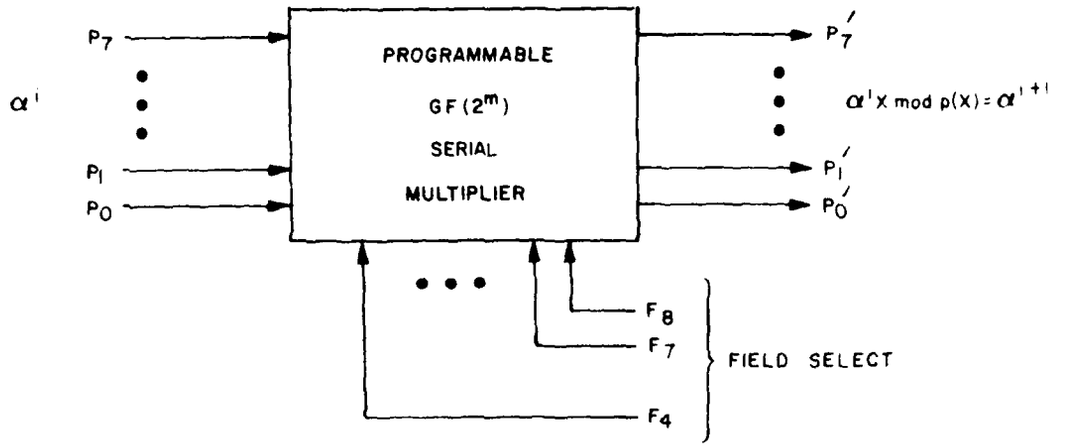


Figure A-7. Sequential GF(2^m) Multiplication Using a Programmable Serial Multiplier



$$P_o' = P_3F_4 + P_4F_5 + P_5F_6 + P_6F_7 + P_7F_8$$

$$P_1' = P_o + P_3F_4 + P_5F_6$$

$$P_2' = P_1 + P_4P_5 + P_7F_8$$

$$P_3' = P_2 + P_6F_7 + P_7F_8$$

$$P_4' = P_3\overline{F}_4 + P_7F_8$$

$$P_5' = P_4(\overline{F}_4+\overline{F}_5)$$

$$P_6' = P_5(F_7+F_8)$$

$$P_7' = P_6F_8$$

Figure A-8. Programmable GF(2^m) Serial Multiplier

input to the serial multiplier, and the symbols $\{P'_0, P'_1, \ldots, P'_7\}$ represent the output of the multiplier. As before, the symbols F_m are flags that are used to indicate the field of operation. For a given input α^j, the serial multiplier calculates

$$\alpha^{j+1} = (\alpha_0^j + \alpha_1^j x + \cdots + \alpha_7^j x^7)x \bmod p(x) \qquad \text{(A-12)}$$

where α^j and α^{j+1} are elements from $GF(2^m)$. Equation (A-12) can be interpreted as shifting the 8-bit representation of the field element α^j and performing modulo p(x) field reduction. The field reduction is implemented in an end-around carry technique that is identical to that used in both the programmable array multiplier and the α^2 multiplier. However, the modular reduction is trivial in the serial multiplier because the product polynomial that is to be reduced is only of degree m. Since p(x) is also of degree m, only one coefficient has to be fed back for each field of operation. The simplicity of this circuit is indicated by the logic equations shown in Figure A-8.

The $GF(2^m)$ serial multiplier operates with sequential data. Many partial products can be "summed" in a binary fashion, and the $GF(2^m)$ serial multiplier can be used to implement associative multiplication

$$\alpha^i (\alpha^0 + \alpha^1 + \alpha^2 + \cdots + \alpha^N) \qquad \text{(A-13)}$$

This type of multiplication is implemented by forming the binary sums (modulo-two) of the terms within the parenthesis and then using the serial multiplier to complete the multiplication. In this manner, N+1 $GF(2^m)$ multipliers can be replaced by one $GF(2^m)$ serial multiplier, in the formation of convolution products.

APPENDIX B

AN EXAMPLE: A (31,15) REED-SOLOMON CODE CONSTRUCTED OVER $GF(2^5)$

In this appendix, the decoding algorithm is illustrated by means of an example of decoding with the (31,15) Reed-Solomon code. A message sequence of length k = 15, over $GF(2^5)$ is given. Encoding is performed by forming a sequence of length-31 with zeros in the first 16 positions and the information symbols in the remaining positions, and then applying the inverse transformation to yield a length-31 codeword, also over $GF(2^5)$. Field elements are represented by powers of a primitive element α.

The representation of $GF(2^5)$ as binary polynomials modulo the irreducible polynomial $x^5 + x^2 + 1$ is:

$0 = 00000$	$\alpha^7 = 10100$	$\alpha^{15} = 11111$	$\alpha^{23} = 01111$
$\alpha^0 = 00001$	$\alpha^8 = 01101$	$\alpha^{16} = 11011$	$\alpha^{24} = 11110$
$\alpha^1 = 00010$	$\alpha^9 = 11010$	$\alpha^{17} = 10011$	$\alpha^{25} = 11001$
$\alpha^2 = 00100$	$\alpha^{10} = 10001$	$\alpha^{18} = 00011$	$\alpha^{26} = 10111$
$\alpha^3 = 01000$	$\alpha^{11} = 00111$	$\alpha^{19} = 00110$	$\alpha^{27} = 01011$
$\alpha^4 = 10000$	$\alpha^{12} = 01110$	$\alpha^{20} = 01100$	$\alpha^{28} = 10110$
$\alpha^5 = 00101$	$\alpha^{13} = 11100$	$\alpha^{21} = 11000$	$\alpha^{29} = 01001$
$\alpha^6 = 01010$	$\alpha^{14} = 11101$	$\alpha^{22} = 10101$	$\alpha^{30} = 10010$

Addition is defined as component-by-component addition modulo 2, and multiplication by addition of exponents modulo 31.

Let the message be the arbitrarily chosen sequence $\{M_i\}$, i = 0, 1, ..., 14, where:

$$M_0 = \alpha^4 \qquad M_8 = \alpha^7$$

$$M_1 = \alpha^{12} \qquad M_9 = \alpha^{15}$$

$$M_2 = \alpha^{28} \qquad M_{10} = \alpha^{24}$$

$$M_3 = \alpha^{29} \qquad M_{11} = \alpha^{1}$$

$$M_4 = \alpha^{17} \qquad M_{12} = \alpha^{6}$$

$$M_5 = \alpha^{3} \qquad M_{13} = \alpha^{14}$$

$$M_6 = \alpha^{30} \qquad M_{14} = \alpha^{20}$$

$$M_7 = \alpha^{10}$$

The padded message sequence $\{A_i\}$ is then

$$A_0 = 0$$

$$A_1 = 0$$

$$\vdots$$

$$A_{15} = 0$$

$$A_{16} = M_0 = \alpha^4$$

$$A_{17} = M_1 = \alpha^{12}$$

$$\vdots$$

$$A_{30} = M_{14} = \alpha^{20}$$

To compute $\{a_i\}$, the inverse transform of the $\{A_i\}$, consider $A(x) = A_0 + A_1 x + \cdots + A_{30} x^{30}$, the corresponding 30th degree polynomial. Then

$$a_i = A(\alpha^{-i}), \qquad i = 0, 1, \ldots, 30$$

To evaluate $A(\alpha^{-i})$, evaluate $t_i(\alpha^{-i})$ where $t_{-i}(x)$ is the residue polynomial which corresponds to division of $A(x)$ by $m_{-i}(x)$, the mini-

mum polynomial of α^{-i}. That is

$$a_i = A(\alpha^{-i}) = t_{-i}(\alpha^{-i}), \qquad i = 0, 1, \ldots, 30 .$$

The seven minimum polynomials of the elements of $GF(2^5)$ over GF(2), and the corresponding $t_i(x)$ are listed in Table B-1.

To illustrate this procedure,

$$\begin{aligned} a_{15} &= A(\alpha^{-15}) = A(\alpha^{16}) \\ &= t_{16}(\alpha^{16}) = t_1(\alpha^{16}) \quad (\text{since } m_{16}(x) = m_1(x)) \\ &= \alpha^{22} + \alpha^{24}(\alpha^{16}) + \alpha^{25}(\alpha^{16})^2 + \alpha^{27}(\alpha^{16})^3 + \alpha^{29}(\alpha^{16})^4 \\ &= \alpha^{22} + \alpha^{9} + \alpha^{29} + \alpha^{28} \\ &= \alpha^{11} \end{aligned}$$

The rest of the inverse transform is determined similarly, producing

$a_0 = \alpha^{22}$	$a_{11} = \alpha^{12}$	$a_{21} = \alpha^{28}$
$a_1 = \alpha^{5}$	$a_{12} = \alpha^{11}$	$a_{22} = \alpha^{27}$
$a_2 = \alpha^{26}$	$a_{13} = \alpha^{22}$	$a_{23} = \alpha^{3}$
$a_3 = \alpha^{18}$	$a_{14} = \alpha^{0}$	$a_{24} = \alpha^{9}$
$a_4 = \alpha^{18}$	$a_{15} = \alpha^{11}$	$a_{25} = \alpha^{19}$
$a_5 = \alpha^{22}$	$a_{16} = \alpha^{26}$	$a_{26} = \alpha^{4}$
$a_6 = \alpha^{21}$	$a_{17} = \alpha^{1}$	$a_{27} = \alpha^{24}$
$a_7 = 0$	$a_{18} = \alpha^{20}$	$a_{28} = \alpha^{11}$
$a_8 = \alpha^{16}$	$a_{19} = \alpha^{28}$	$a_{29} = 0$
$a_9 = \alpha^{23}$	$a_{20} = \alpha^{17}$	$a_{30} = \alpha^{6}$
$a_{10} = \alpha^{30}$		

Table B-I

Minimal Polynomials of $GF(2^5)$ Over GF(2) and Remainder Polynomials Corresponding to $A(x)/m_i(x)$ and $r(x)/m_i(x)$

$m_i(x)$	$t_i(x)$	$u_i(x)$
$m_0(x) = 1 + x$	α^{22}	α^{28}
$m_1(x) = 1 + x^2 + x^5$	$\alpha^{22} + \alpha^{24}x + \alpha^{25}x^2 + \alpha^{12}x^3 + \alpha^{26}x^4$	$\alpha^{14} + \alpha^{20} + \alpha^{7}x^2 +$
$m_3(x) = 1 + x^2 + x^3 + x^4 + x^5$	$\alpha^{0} + \alpha^{21}x^2 + \alpha^{7}x^3 + \alpha^{16}x^4$	$\alpha^{10} + \alpha^{3}x +$
$m_5(x) = 1 + x + x^2 + x^4 + x^5$	$\alpha^{24} + \alpha^{12}x + \alpha^{24}x^2 + \alpha^{18}x^3 + \alpha^{21}x^4$	$\alpha^{20} + \alpha^{11}x + \alpha^{11}x^2 + \alpha^{24}x^3 + \alpha^{2}x^4$
$m_7(x) = 1 + x + x^2 + x^3 + x^5$	$\alpha^{6} + \alpha^{10}x + \alpha^{28}x^2 + \alpha^{12}x^3$	$\alpha^{29} + \alpha^{28}x + \alpha^{23}x^2 + \alpha^{1}x^3 + \alpha^{26}x^4$
$m_{11}(x) = 1 + x + x^3 + x^4 + x^5$	$\alpha^{23} + \alpha^{5}x + \alpha^{27}x^2 + \alpha^{20}x^3 + \alpha^{18}x^4$	$\alpha^{15} + \alpha^{23}x + \alpha^{23}x^2 + \alpha^{15}x^3 + \alpha^{23}x^4$
$m_{15}(x) = 1 + x^3 + x^5$	$\alpha^{14} + \alpha^{5}x + \alpha^{21}x^2 + \alpha^{7}x^3 + \alpha^{25}x^4$	$\alpha^{25} + \alpha^{6}x + \alpha^{7}x^2 + \alpha^{25}x^3 + \alpha^{18}x^4$

$m_i(x)$ = i^{th} minimum polynomial of $GF(2^5)$ over GF(2).

$t_i(x)$ = remainder polynomial corresponding to division of A(x) by $m_i(x)$.

$u_i(x)$ = remainder polynomial corresponding to division of r(x) by $m_i(x)$.

This is the transmitted message.

Suppose the following t = 5 errors and r = 6 erasures are introduced, such that

Errors	Erasure Locations (known)
a_1: $\alpha^5 \rightarrow \alpha^{14}$	a_6 : α^6
a_{14}: $\alpha^0 \rightarrow \alpha^{18}$	a_7 : α^7
a_{18}: $\alpha^{20} \rightarrow \alpha^{22}$	a_8 : α^8
a_{23}: $\alpha^3 \rightarrow \alpha^{11}$	a_9 : α^9
a_{29}: $0 \rightarrow \alpha^9$	a_{10}: α^{10}
	a_{11}: α^{11}

where a_i: $\alpha^j \rightarrow \alpha^k$ means that a_i is changed from α^j to α^k. The erasures are consecutive to simulate a burst. The received sequence, $\{r_i\}$, is

$r_0 = \alpha^{22}$	$r_{12} = \alpha^{11}$	$r_{21} = \alpha^{28}$
$r_1 = \alpha^{14}$ (error)	$r_{13} = \alpha^{22}$	$r_{22} = \alpha^{27}$
$r_2 = \alpha^{26}$	$r_{14} = \alpha^{18}$ (error)	$r_{23} = \alpha^{11}$ (error)
$r_3 = \alpha^{13}$	$r_{15} = \alpha^{11}$	$r_{24} = \alpha^{9}$
$r_4 = \alpha^{18}$	$r_{16} = \alpha^{26}$	$r_{25} = \alpha^{19}$
$r_5 = \alpha^{22}$	$r_{17} = \alpha^{1}$	$r_{26} = \alpha^{4}$
$r_6 = \alpha^{21}$	$r_{18} = \alpha^{22}$ (error)	$r_{27} = \alpha^{24}$
$r_7 = 0$	$r_{19} = \alpha^{28}$	$r_{28} = \alpha^{11}$
$r_8 = \alpha^{16}$ (erasures)	$r_{20} = \alpha^{17}$	$r_{29} = \alpha^{9}$ (error)
$r_9 = \alpha^{23}$		$r_{30} = \alpha^{6}$
$r_{10} = \alpha^{30}$		
$r_{11} = \alpha^{12}$		

where r_6, r_7, ..., r_{11} are received correctly in this case but with enough uncertainty to warrant labeling them erasures. Altering them changes neither the algorithm nor the decoded message.

Decoding now begins. Since the bound $2t + \nu \leq n-k$ is satisfied, the message is recovered completely. A flowchart of the algorithm is shown in Figure B-1. There,

$\nu = 6$ = number of erasures

$\{r_i\}$ $i = 0, 1, \ldots, 31$

$\{\alpha^{i_\ell}\}_{\ell = 0, 1, \ldots, 5}$ = erasure locations with

$i_0 = 11$, $i_1 = 10$, ... and $i_5 = 16$.

The algorithm first generates the erasure polynomial $\Lambda^{(\nu-1)}(x) = \Lambda^{(5)}(x)$. The dummy variable, ν', counts the number of erasures by decreasing from 5 to 0, after which time the generation of the errata-locator polynomial begins and a different path is followed in the flowchart.

Table B-II displays the result of all iterations up to $N = n-k-1 = 15$ after which the errata locator polynomial, $\Lambda^{(15)}(x)$, is synthesized. Each line represents one iteration and is filled out from left to right.

The first step in decoding is to compute R_N, the Nth term of $\{R_i\}$, the forward transform of the received sequence $\{r_i\}$.

Here,

$$R_i = r(\alpha^i) = u_i(\alpha^i),\ i = 0, 1, \ldots, 30$$

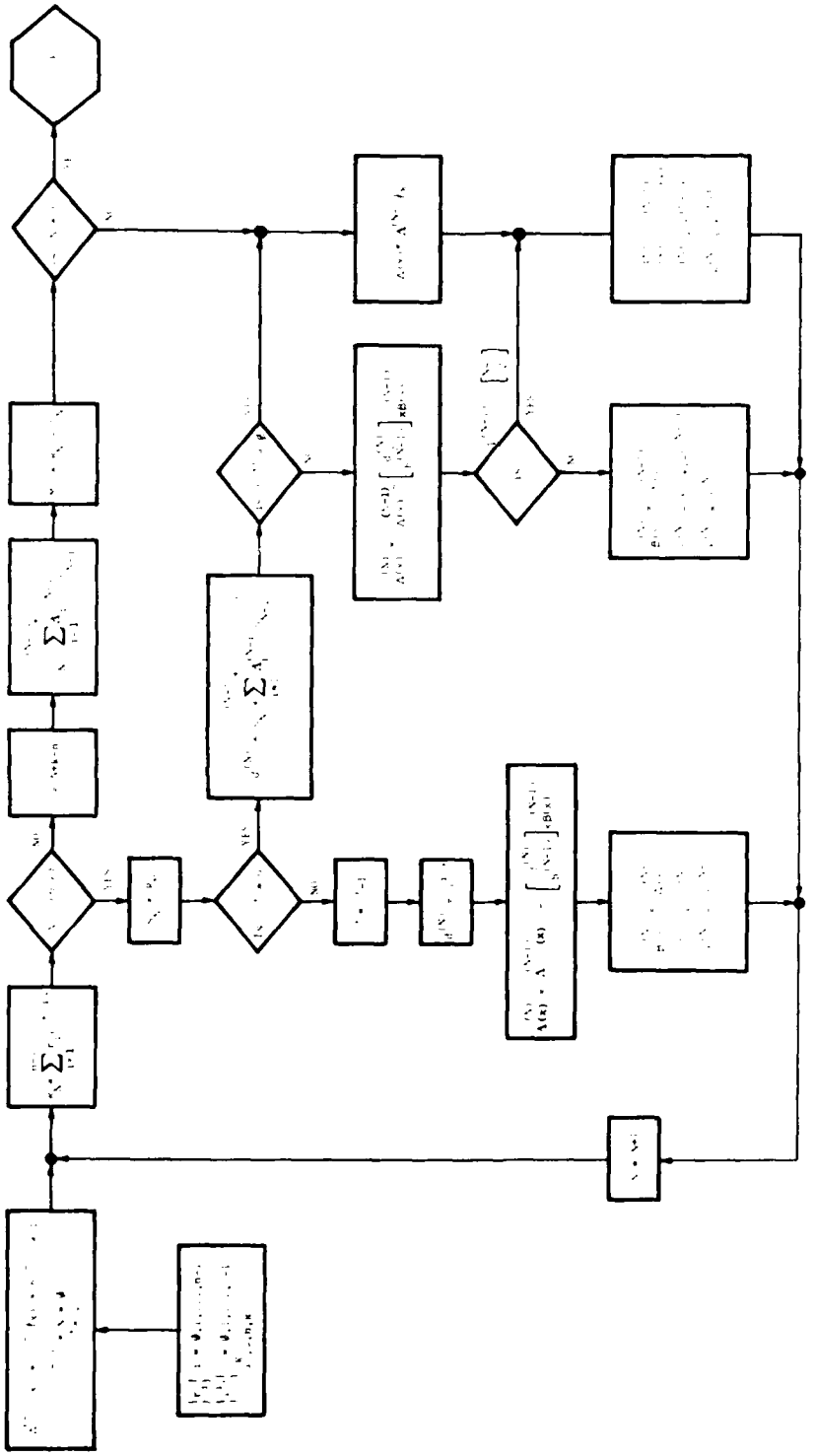


Figure B-1. Flowchart of Transform Decoding Algorithm

Table B-II

Result of First n-k = 16 Iterations

N	r_N	S_N	$\Delta^{(N)}$	$d^{(N)}$	$\Lambda^{(N)}(x)$ Coefficients												$B^{(N)}(x)$ Coefficients												$L^{(N)}$	$b^{(N)}$
					1	x	x^2	x^3	x^4	x^5	x^6	x^7	x^8	x^9	x^{10}	x^{11}	1	x	x^2	x^3	x^4	x^5	x^6	x^7	x^8	x^9	x^{10}			
-1			6		1												1												0	1
0	α^{29}	α^{28}	5	α^{6}	1	α^{6}											1	α^{6}											0	1
1	α^{23}	α^{23}	4	α^{7}	1	α^{24}	α^{13}										1	α^{24}	α^{13}										0	1
2	α^{5}	α^{5}	3	α^{8}	1	α^{17}	α^{24}	α^{21}									1	α^{17}	α^{24}	α^{21}									0	1
3	α^{25}	α^{25}	2	α^{9}	1	α^{29}	α^{29}	α^{13}	α^{30}								1	α^{29}	α^{29}	α^{13}	α^{30}								0	1
4	α^{29}	α^{29}	1	α^{10}	1	α^{21}	α^{2}	α^{10}	α^{14}	α^{9}							1	α^{21}	α^{2}	α^{10}	α^{14}	α^{9}							0	1
5	α^{26}	α^{26}	0	α^{11}	1	α^{15}	α^{19}	α^{8}	α^{5}	α^{18}	α^{20}						1	α^{15}	α^{19}	α^{8}	α^{5}	α^{18}	α^{20}						0	1
6	α^{1}	α^{1}	0	α^{26}	1	α^{1}	α^{26}	α^{9}	α^{8}	α^{2}	α^{4}	α^{15}					1	α^{15}	α^{19}	α^{8}	α^{5}	α^{18}	α^{20}						1	α^{26}
7	α^{29}	α^{29}	0	α^{12}	1	α^{16}	α^{22}	α^{22}	α^{7}	α^{26}	0	α^{22}					0	α^{1}	α^{15}	α^{19}	α^{8}	α^{5}	α^{18}	α^{20}					1	α^{26}
8	α^{24}	α^{24}	0	α^{27}	1	α^{16}	α^{26}	α^{22}	α^{21}	α^{8}	α^{6}	α^{27}	α^{21}				1	α^{16}	α^{22}	α^{22}	α^{7}	α^{26}	0	α^{22}					2	α^{27}
9	α^{10}	α^{10}	0	α^{4}	1	α^{28}	α^{29}	α^{13}	α^{6}	α^{30}	α^{1}	α^{17}	α^{6}				0	1	α^{16}	α^{22}	α^{22}	α^{7}	α^{26}	0	α^{22}				2	α^{27}
10	α^{24}	α^{24}	0	α^{2}	1	α^{28}	α^{28}	α^{29}	α^{13}	α^{6}	α^{30}	α^{1}	α^{17}	α^{6}			1	α^{28}	α^{29}	α^{13}	α^{6}	α^{30}	α^{1}	α^{17}	α^{6}				3	α^{2}
11	α^{24}	α^{24}	0	α^{25}	1	α^{25}	α^{27}	α^{23}	α^{19}	α^{25}	α^{9}	α^{27}	α^{23}	α^{8}	α^{15}	0	0	1	α^{28}	α^{29}	α^{13}	α^{6}	α^{30}	α^{1}	α^{17}	α^{6}			3	α^{2}
12	α^{7}	α^{7}		0	1	α^{25}	α^{25}	α^{19}	α^{25}	α^{9}	α^{27}	α^{23}	α^{8}	α^{15}	0		0	0	1	α^{28}	α^{29}	α^{13}	α^{6}	α^{30}	α^{1}	α^{17}	α^{6}			3	α^{2}
13	α^{6}	α^{6}	0	α^{18}	1	α^{25}	α^{25}	α^{19}	α^{25}	α^{8}	α^{7}	α^{24}	α^{8}	α^{15}	0		1	α^{25}	α^{25}	α^{19}	α^{25}	α^{8}	α^{7}	α^{24}	α^{8}	α^{15}			4	α^{18}
14				α^{7}	1	α^{24}	α^{7}	α^{8}	α^{7}	α^{9}	α^{6}	α^{25}	α^{24}	α^{15}			1	α^{24}	α^{7}	α^{8}	α^{7}	α^{9}	α^{6}	α^{25}	α^{24}	α^{15}			4	α^{18}
15	α^{14}	α^{14}	1	α^{19}	1	α^{7}	α^{14}	α^{15}	α^{14}	α^{16}	α^{13}	α^{31}	α^{30}	α^{22}			1	α^{7}	α^{14}	α^{15}	α^{14}	α^{16}	α^{13}	α^{31}	α^{30}	α^{22}			5	α^{19}

Note: $\Lambda^{(15)}(x)$ is the errata locator polynomial.

where $r(x) = r_0 + r_1 x + \cdots + r_{30} x^{30}$ is the 30th degree polynomial which corresponds to $\{r_i\}$, and $u_i(x)$ is the remainder polynomial which results from division of $r(x)$ by $m_i(x)$. (The remainders, $u_i(x)$, are also listed in Table B-I). The forward transform, $\{R_i\}$, of the received sequence is

$R_0 = \alpha^{28}$	$R_8 = \alpha^{24}$	$R_{16} = \alpha^{7}$	$R_{24} = \alpha^{10}$
$R_1 = \alpha^{23}$	$R_9 = \alpha^{10}$	$R_{17} = \alpha^{9}$	$R_{25} = \alpha^{1}$
$R_2 = \alpha^{5}$	$R_{10} = \alpha^{14}$	$R_{18} = \alpha^{27}$	$R_{26} = \alpha^{29}$
$R_3 = \alpha^{25}$	$R_{11} = \alpha^{13}$	$R_{19} = \alpha^{10}$	$R_{27} = \alpha^{26}$
$R_4 = \alpha^{29}$	$R_{12} = \alpha^{4}$	$R_{20} = \alpha^{21}$	$R_{28} = \alpha^{10}$
$R_5 = \alpha^{26}$	$R_{13} = \alpha^{6}$	$R_{21} = \alpha^{8}$	$R_{29} = \alpha^{28}$
$R_6 = \alpha^{1}$	$R_{14} = 0$	$R_{22} = \alpha^{1}$	$R_{30} = \alpha^{10}$
$R_7 = \alpha^{29}$	$R_{15} = \alpha^{14}$	$R_{23} = \alpha^{7}$	

Since $GF(2^5)$ is of characteristic 2, addition and subtraction are both given by mod 2 addition of the 5-tuples. The first few computations are:

$$\Lambda^{(0)}(x) = \Lambda^{(-1)}(x) - \frac{d^{(0)}}{b^{(-1)}} \times \beta^{(-1)}(x)$$

$$= 1 - \left[\frac{\alpha^6}{1}\right] x \left(1\right) = 1 - \alpha^6 x = 1 + \alpha^6 x$$

$$\Lambda^{(1)}(x) = \Lambda^{(0)}(x) - \left[\frac{d^{(1)}}{b^{(0)}}\right] x \beta^{(0)}(x)$$

$$= (1 + \alpha^6 x) - \left[\frac{\alpha^7}{1}\right] x (1 + \alpha^6 x)$$

$$= 1 + (\alpha^6 + \alpha^7)\, x + \alpha^{13}\, x^2$$

$$= 1 + \alpha^{24}\, x + \alpha^{13}\, x^2$$

$$\vdots$$

$$\Lambda^{(5)}(x) = \Lambda^4(x) - \left[\frac{\alpha^{11}}{1}\right] x\, \beta^4(x)$$

$$= 1 + (\alpha^{21} + \alpha^{11})\, x + (\alpha^2 + \alpha^1)\, x^2 + (\alpha^{10} + \alpha^{13})\, x^3$$

$$+ (\alpha^{14} + \alpha^{21})\, x^4 + (\alpha^9 + \alpha^{25})\, x^5 + \alpha^{20}\, x^6$$

$$= 1 + \alpha^{15}\, x + \alpha^{19}\, x^2 + \alpha^8\, x^3 + \alpha^5\, x^4 + \alpha^{18}\, x^5 + \alpha^{20}\, x^6.$$

When $N = 6$, $\nu' = 0$, and the algorithm branches with the computation

$$d^{(6)} = S_6 + \sum_{i=1}^{L^{(5)}+6} \Lambda_i^{(5)}\, S_{6-i}$$

$$= \alpha^1 + \alpha^{15} \cdot \alpha^{26} + \alpha^{19} \cdot \alpha^{29} + \alpha^8 \cdot \alpha^{25} + \alpha^5 \cdot \alpha^5$$
$$+ \alpha^{18} \cdot \alpha^{23} + \alpha^{20} \cdot \alpha^{28}$$

$$= \alpha^1 + \alpha^{10} + \alpha^{17} + \alpha^2 + \alpha^{10} + \alpha^{10} + \alpha^{27}$$

$$= \alpha^{26}.$$

Since $d^{(6)} \neq 0$, the "NO" path is followed and

$$\Lambda^{(6)}(x) = \Lambda^{(5)}(x) - \left[\frac{\alpha^{26}}{1}\right] x\ \beta^{(5)}(x)$$

$$= 1 + (\alpha^{15} + \alpha^{26})\ x + (\alpha^{19} + \alpha^{10})\ x^2 + (\alpha^{8} + \alpha^{14})\ x^3$$
$$+ (\alpha^{5} + \alpha^{3})\ x^4 + (\alpha^{16} + \alpha^{0})\ x^5$$

$$= 1 + \alpha^{3}\ x + \alpha^{26}\ a^2 + \alpha^{4}\ x^3 + \alpha^{8}\ x^4 + \alpha^{1}\ x^5$$
$$+ \alpha^{4}\ x^6 + \alpha^{15}\ x^7.$$

Now $L^{(5)} = 0 \leq \frac{6-6}{2} = 0$, so the "NO" path is followed. This process continues until errata locator polynomial $\Lambda^{(15)}(x)$ is computed. The iteration at N = 15 is:

$$\Lambda^{(15)}(x) = \Lambda^{(14)}(x) - \left[\frac{\alpha^{19}}{\alpha^{18}}\right] x\ \beta^{(14)}(x)$$

$$= \Lambda^{(14)}(x) - \alpha^{1}\ x\ \beta^{(14)}(x)$$

$$= 1 + (\alpha^{5} + 0)\ x + (\alpha^{16} + \alpha^{1})\ x^2 + (\alpha^{23} + \alpha^{26})\ x^3$$
$$+ (\alpha^{6} + \alpha^{24})\ x^4 + (\alpha^{2} + \alpha^{11})\ x^5 + (\alpha^{9} + \alpha^{26})$$
$$x^6 + (0 + \alpha^{9})\ x^7 + (\alpha^{6} + \alpha^{28})\ x^8 + (\alpha^{15} + \alpha^{24})$$
$$x^9 + (\alpha^{30} + \alpha^{5})\ x^{10} + (\alpha^{22} + \alpha^{16})\ x^{11}$$

$$= 1 + \alpha^{5}\ x + \alpha^{25}\ x^2 + \alpha^{21}\ x^3 + \alpha^{7}\ x^4 + \alpha^{18}\ x^5 +$$
$$\alpha^{8} + x^6 + \alpha^{9}\ x^7 + \alpha^{13}\ x^8 + \alpha^{0}\ x^9 + \alpha^{26}\ x^{10}$$
$$+ \alpha^{12}\ x^{11}$$

At this point N is incremented by 1 to $N = 16 \geq n-k = 16$, so that the algorithm branches to generate the error sequence $(E_0, E_1, \ldots, E_{14}) = (S_{16}, S_{17}, \ldots, S_{30})$, and then the decoded message $\{\hat{M}_i\} = \{\hat{M}_i = R_{16+i} + E_{16+i}\}$. To illustrate:

$$\ell = N + k - n = 16 + 15 - 31 = 0$$

$$S_{16} = \sum_{i=1}^{11} \Lambda_i^{(15)} S_{16-i}$$

$$= \alpha^5 \cdot \alpha^{14} + \alpha^{25} \cdot 0 + \alpha^{21} \cdot \alpha^6 + \alpha^7 \cdot \alpha^4 + \alpha^{18} \cdot \alpha^{13} + \alpha^8 \cdot \alpha^{14} + \alpha^9 \cdot \alpha^{10} + \alpha^{13} \cdot \alpha^{24} + \alpha^0 \cdot \alpha^{29} + \alpha^{26} \cdot \alpha^1 + \alpha^{12} \cdot \alpha^{26}$$

$$= \alpha^{19} + \alpha^{27} + \alpha^{11} + \alpha^0 + \alpha^{22} + \alpha^{19} + \alpha^6 + \alpha^{29} + \alpha^{27} + \alpha^7$$

$$= \alpha^2$$

so $\hat{M}_\ell = \hat{M}_0 = R_{16} - S_{16} = \alpha^7 - \alpha^2$ which is M_0, the first symbol of the transmitted message.

The algorithm continues until M_i, i=0, 1, ..., 14 are computed. They are:

$\hat{M}_0 = \alpha^4$
$\hat{M}_1 = \alpha^{12}$
$\hat{M}_2 = \alpha^{28}$
$\hat{M}_3 = \alpha^{29}$
$\hat{M}_4 = \alpha^{17}$
$\hat{M}_5 = \alpha^3$
$\hat{M}_6 = \alpha^{30}$
$\hat{M}_7 \quad \alpha^{10}$
$\hat{M}_8 = \alpha^7$
$\hat{M}_9 = \alpha^{15}$
$\hat{M}_{10} = \alpha^{24}$
$\hat{M}_{11} = \alpha^1$
$\hat{M}_{12} = \alpha^6$
$\hat{M}_{13} = \alpha^{14}$
$\hat{M}_{14} = \alpha^{20}$

REFERENCES

1. Peterson, W., Weldon, J., Error Correcting Codes, 2nd Ed., MIT Press, Cambridge, MA, 1977.

2. Haggarty, R.D., Palo, E. A., Carhoun, D. O. and Meehan, S. J., High Speed Signal Processing for Error Correction Coding," presented at IEEE International Symposium on Circuits and Systems, Tokyo, Japan, 1979.

3. Hamalainen, J. R. and Skoog, E. N., "Error Correcting Coding with NMOS Microprocessors: A 6800-based (7,3) Reed-Solomon Decoder," ESD-TR-79-125, Vol. II, AD A073088, September 1978.

4. Carhoun, D. O., Johnson, B. L. and Meehan, S. J., "Transform Decoding of Reed-Solomon Codes, Volume I: Algorithm and Signal Processing Structure," ESD-TR-82-403, Vol. I, November 1982. (In Process)

5. Blahut, R. E., "Transform Techniques for Error Control Codes," IBM J. Res. Development, Vol. 23, No. 3, May 1979.

6. Berlekamp, E. R., Algebraic Coding Theory, McGraw-Hill, New York, 1968.

7. Massey, J. L., "Shift-Register Synthesis and BCH Decoding," IEEE Transaction on Information Theory, Vol. IT, No. 1, pp. 122-127, January 1969.

8. Carhoun, D. O., Johnson, B. L., and Meehan, S. J., "An Architectural Design for a Fast Number Theoretic Transformer," presented at the IEEE Custom Integrated Circuits Conference, Rochester, NY, May 1981.

9. Carhoun, D. O., Johnson, B. L. and Meehan, S. J., "VLSI Architectural Design for a Reed-Solomon Transform Decoder," presented at the IEEE International Symposium on Circuits and Systems, Chicago, IL, April 1981.

10. Mead, C. A. and Conway, L. A., Introduction to VLSI Systems, Addison-Wesley, 1979.


ME
8